Neural Program Generation Modulo Static Analysis

Rohan Mukherjee
Rice University

Yeming Wen
UT Austin

Dipak Chaudhari
UT Austin

Thomas W. Reps
University of Wisconsin

Swarat Chaudhuri
UT Austin

Chris Jermaine
Rice University

Abstract

State-of-the-art neural models of source code tend to be evaluated on the generation of individual expressions and lines of code, and commonly fail on long-horizon tasks such as the generation of entire method bodies. We propose to address this deficiency using weak supervision from a static program analyzer. Our neurosymbolic method allows a deep generative model to symbolically compute, using calls to a static-analysis tool, long-distance semantic relationships in the code that it has already generated. During training, the model observes these relationships and learns to generate programs conditioned on them. We apply our approach to the problem of generating entire Java methods given the remainder of the class that contains the method. Our experiments show that the approach substantially outperforms state-of-the-art transformers and a model that explicitly tries to learn program semantics on this task, both in terms of producing programs free of basic semantic errors and in terms of syntactically matching the ground truth.

1 Introduction

Neural models of source code have received much attention in the recent past [38, 9, 26, 23, 30, 36, 16, 24]. However, these models have a basic weakness: while they frequently excel at generating individual expressions or lines of code, they do not do so well when tasked with synthesizing larger code blocks. For example, as we show later in this paper, state-of-the-art transformer models [8, 6, 24] can generate code with elementary semantic errors, such as uninitialized variables and type-incorrect expressions, when asked to generate *method bodies*, as opposed to single lines. Even in terms of syntactic accuracy measures, the quality of the code that transformers produce on such "long-horizon" tasks can be far removed from the ground truth.

The root cause of these issues, we believe, is that current neural models of code treat programs as text rather than artifacts that are constructed following a *semantics*. In principle, a model could learn semantics from syntax given enough data. In practice, such learning is difficult for complex, general-purpose languages.

In this paper, we propose to address this challenge through an alternative *neurosymbolic* approach. Our main observation is that symbolic methods—specifically, static program analysis—can extract deep semantic relationships between far-removed parts of a program. However, these relationships are not apparent at the level of syntax, and it is difficult for even large neural networks to learn them automatically. Driven by this observation, we use a static-analysis tool as a *weak supervisor* for a deep model of code. During generation, our model invokes this static analyzer to compute a set of semantic facts about the code generated so far. The distribution over the model's next generation actions is conditioned on these facts.

We concretely develop our approach by extending the classic formalism of *attribute grammars* [20]. Attribute grammars are like context-free grammars but allow rules to carry symbolic *attributes* of the

35th Conference on Neural Information Processing Systems (NeurIPS 2021).

context in which a rule is fired. In our model, called *Neurosymbolic Attribute Grammars* (NSGs), the context is an incomplete program, and rules are fired to replace a nonterminal (a stand-in for unknown code) in this program. The attributes are semantic relationships (for example, symbol tables) computed using static analysis. The neural part of the model represents a probability distribution over the rules of the grammar conditioned on the attributes. During generation, the model repeatedly samples from this distribution while simultaneously computing the attributes of the generated code.

We evaluate our approach in the task of generating the entire body of a Java method given the rest of the class in which the method occurs. We consider a large corpus of curated Java programs, over a large vocabulary of API methods and types.Using this corpus, we train an NSG whose attributes, among other things, track the state of the symbol table and the types of arguments and return values of invoked methods at various points of a program, and whose neural component is a basic tree LSTM. We compare this model against several recent models: fine-tuned versions of two GPT-NEO [6] transformers and the CODEGPT [24] transformer, OpenAI's CODEX system [8] (used in a zero-shot manner), and a GNN-based method for program encoding [7]. Some of these models are multiple orders of magnitude larger than our NSG model. Our experiments show that the NSG model reliably outperforms all of the baselines on our task, both in terms of producing programs free of semantic errors and in terms of matching the ground truth syntactically.

In summary, this paper makes three contributions:

- We present a new approach to the generative modeling of source code that uses a static-analysis tool as a weak supervisor.
- We embody this approach in the specific form of *neurosymbolic attribute grammars* (NSGs).
- We evaluate the NSG approach on the long-horizon task of generating entire Java method bodies, and show that it significantly outperforms several larger, state-of-the-art transformer models.

2 Conditional Program Generation

We start by stating our problem, known as conditional program generation (CPG) [26]. We imagine a joint distribution $\mathcal{D}(X, Y)$, where X ranges over *specifications* of program-generation problems and Y ranges over programs. The probability $\mathcal{D}(X = \mathsf{X}, Y = \mathsf{Y})$ is high when Y is a solution to X. Also, we consider a family of distributions $\mathcal{P}_\theta(Y|X = \mathsf{X})$, parameterized by θ, that we might want to learn. *Learning to conditionally generate* programs amounts to finding parameters θ that minimize the prediction error $\mathbf{E}_{(\mathsf{X},\mathsf{Y})\sim\mathcal{D}}[\delta(\mathcal{P}_\theta(\mathsf{X}|\mathsf{Y}), \mathsf{Y})]$, where δ is a suitable distance function between programs.

Specifications and distances between programs can be defined in many ways. In our experiments, the goal is to generate Java method bodies. A specification is an *evidence set* that contains information—e.g., method names, types of variables and methods—about the class in which the method lies. We define $\delta(\mathsf{Y}_1, \mathsf{Y}_2)$ to be a large number if Y_1 or Y_2 violates one of several language-level invariants (e.g., type-safety, initialization of variables before use) that we require programs to satisfy. When both programs satisfy the invariants, $\delta(\mathsf{Y}_1, \mathsf{Y}_2)$ measures the textual dissimilarity between the two programs.

(a)

```
public class FileUtil{
  String err;
  public int read(File f){...}

  /* write lines to file */
  public void write(
    File f, String str){??}}
```

(b)

```
void write(File f, String str){
  try {
    FileWriter var_0;
    var_0 = new FileWriter(f);
    var_0.write(str);
  } catch(IOException var_0) {
    var_0.printStackTrace();
    System.out.println( ARG ); }
  return; }
```

Figure 1: (a) An instance of conditional program generation. (b) A top completion of the `write` method, generated using an NSG. `ARG` stands for a string literal.

Note that CPG is a much more challenging task than the well-studied next-token-prediction task [24, 7]. The goal is to predict long sequences of tokens (e.g., an entire method body). Also, X is a (possibly imprecise) specification of the code to generate, not just a sequence of tokens we are trying to complete by, say, choosing the correct method to call for a variable.

Example. Fig. 1-(a) illustrates the kind of task that we target. Here, we are given a class with a missing `write` method. The specification X includes: (i) the class name `FileUtil`; (ii) the

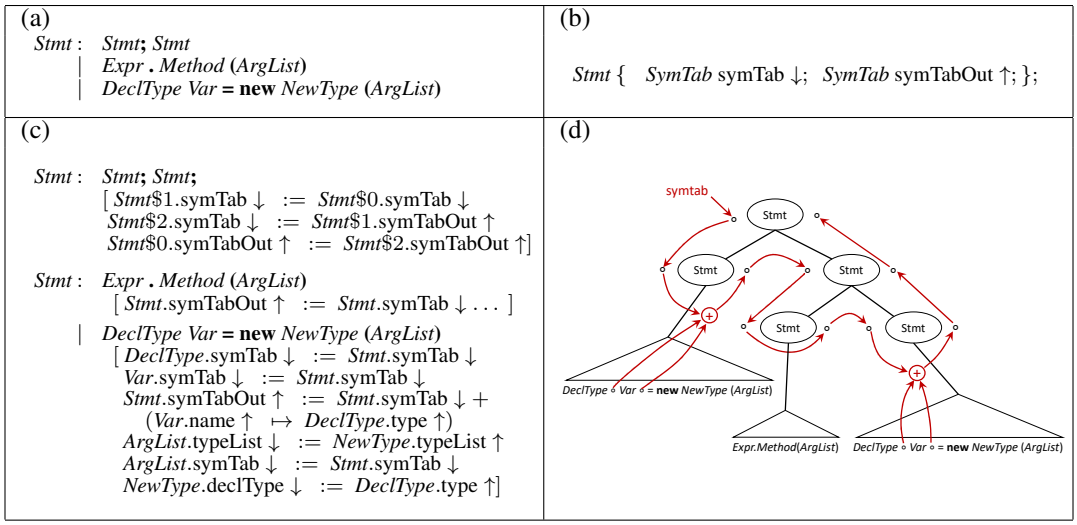

Figure 2: (a) A basic context-free grammar. (b) Attributes of the *Stmt* nonterminal. (c) Attribute equations for the productions (the parts of the equations denoted by "…" are elided). (d) An attributed tree, illustrating left-to-right threading of attributes.

type `String` of the class variable `err`; (iii) information about complete methods within the class (including the methods' return types and formal-parameter types and names, and sequences of API calls made within such methods); (iv) information about the method with missing code (`write`), including its name, formal parameters, and JavaDoc comments for the method with missing code (e.g., "write lines to file"). Our objective on this input is to generate automatically a non-buggy, natural completion of `write`, without any provided, partial implementation of the method.

To understand the challenges in this task, consider a completion that starts by: (i) declaring a local variable `var_0`; and (ii) invoking the constructor for `FileWriter` and storing the result in `var_0`. A proper implementation of these two steps must ensure that `var_0` is of type `FileWriter`. Also, the first argument to the constructor of `FileWriter` must be of type `File` (or a subtype of `File`). As we show in Sec. 5, it is hard for state-of-the-art neural models to learn to satisfy these rules.

In contrast, in our approach, the generator has access to a set of semantic attributes computed via static analysis. These attributes include a *symbol table* mapping in-scope variables to their types.

Suppose that during training we are given the following line of code: "`var_0 = new FileWriter(f, true)`". Our model's symbol table includes the names `var_0` and `f` and their types. The grammar is also able to compute the type of the first argument in the invoked constructor for `FileWriter`. Consequently, the model can observe that the type of `f` is listed as `File` in the symbol table, and that `f` is the first argument to the `FileWriter` constructor. With a few observations like these, the model can learn that the first argument of "`new FileWriter`" tends to be of type `File` (or a subtype). During generation, the model uses this knowledge, locating a variable of the correct type in the symbol table each time it constructs a `FileWriter`.

Fig. 1-(b) shows a top completion of `write` generated by our NSG implementation. Note that all variables in this code are initialized before use, and that all operations are type-safe. Also, the name `var_0` is reused between the **try** and the **catch** blocks. Such reuse is possible because the symbol table carries information about the scopes to which different names belong. Finally, as we will see in Sec. 5, the extra information provided by the static analyzer can also help with accuracy in terms of syntactic matches with the ground truth.

3 Static Analysis with Attribute Grammars

As mentioned in Sec. 1, we develop our approach as an extension of the classic attribute grammar (AG) framework [20]. Now we give some background on static analysis using AGs. In the next section, we show how to use AGs to weakly supervise a neural program generator.

An AG extends a traditional context-free grammar (CFG) [18] by attaching a set of *attributes* to each terminal or nonterminal symbol of the grammar and by using a set of *attribute equations* to propagate attribute values through syntax trees. The attributes of a symbol S can be divided into *inherited* attributes and *synthesized* attributes, which we suffix by $\downarrow$ and $\uparrow$, respectively. Inherited attributes transfer information from parent to child, or from a node to itself. Synthesized attributes transfer information from child to parent, from a node to a sibling, or from a node to itself. We assume that the terminal symbols of the grammar have no synthesized attributes and that the root symbol of the grammar has a special set of inherited attributes, known as the *initial attributes*.

The *output* attributes of a production $S \to S_1, \ldots, S_k$ consist of the synthesized-attribute occurrences of the nonterminal S, plus the inherited-attribute occurrences of all of the S_i's. The *input* attributes are the inherited-attribute occurrences of S, plus the synthesized-attribute occurrences of the S_i's. The grammar's attribute equations relate the input and output attributes of a node in terms of the attributes of its parent, children, and left sibling in the syntax tree that the grammar generates.

Example. Consider the simple CFG in Fig. 2-(a). The nonterminal *Stmt* stands for *program statements*. The grammar says that a statement can either be a sequential composition of statements, a method call, or a variable declaration. A natural AG extension of this CFG tracks symbol tables, which allow easy lookup of all variables in scope.

Specifically, the grammar associates two symbol-table-valued attributes, symTab $\downarrow$ and symTabOut $\uparrow$, with *Stmt* (Fig. 2-(b)). The attributes are propagated following the equations in Fig. 2-(c). In these equations, we distinguish between the three different occurrences of nonterminal "*Stmt*" via the symbols "*Stmt*\$0," "*Stmt*\$1," and "*Stmt*\$2." where the numbers denote the leftmost occurrence, the next-to-leftmost occurrence, etc. In this case, the leftmost occurrence is the left-hand-side occurrence.

For concreteness, let us consider the attribute equations for the production for sequential composition in the grammar. Here, the inherited attribute of *Stmt*\$0 gets passed "down" the syntax tree as an inherited attribute of *Stmt*\$1. The synthesized attribute received at *Stmt*\$1 is passed to *Stmt*\$2 as an inherited attribute. More generally, the attribute equations define a left-to-right information flow through the syntax tree, as illustrated in Fig. 2-(d).

4 Neurosymbolic Attribute Grammars

Now we introduce the model of neurosymbolic attribute grammars (NSGs). Our goal is to learn a distribution $\mathcal{P}(Y|\mathsf{X})$, where Y is a random variable whose domain is all possible programs (concretely, Java method bodies) and X is a specification of a program-generation problem (concretely, an *evidence set* made up of useful information extracted symbolically from the method's context and then encoded using a neural network). Attributes containing the results of a symbolic, static analysis are available to the neural network implementing this distribution. This weak supervision allows the network to mimic more accurately the long-range dependencies present in real code-bases.

The Underlying Model. The idea of weak supervision using a static analyzer could be developed on top of many different kinds of models. Here, we develop the idea on top of a model from Murali et al. [26]. This model uses a latent variable Z to represent the *true user intent* behind the incomplete or ambiguous evidence set Y. We then have $\mathcal{P}(\mathsf{Y}|\mathsf{X}) = \int_{\mathsf{Z}} \mathcal{P}(\mathsf{Z}|\mathsf{X})\mathcal{P}(\mathsf{Y}|\mathsf{Z})d\mathsf{Z}$. To define the distribution $\mathcal{P}(Z|\mathsf{X})$, we assume that the evidence set has data of a fixed number of *types*—e.g., method names, formal parameters, and Javadoc comments.

The j^{th} type of evidence has a neural encoder f_j. An individual piece of evidence X is either encoded as a single vector or as a set of vectors with no particular ordering. For example, our implementation encodes Javadoc comments as vectors using LSTMs, and each member of a set of formal parameters using a basic feedforward network. Let $\mathsf{X}_{j,k}$ refer to the k^{th} instance of the j^{th} kind of evidence in X. Assume a Normal prior on Z, and let $\mathcal{P}(\mathsf{X}|\mathsf{Z}) = \prod_{j,k} \mathcal{N}\left(f_j(\mathsf{X}_{j,k}) \mid \mathsf{Z}, \mathbf{I}\sigma_j^2\right)$. Assume that the encoding of each type of evidence is sampled from a Normal centered at Z. If f is 1-1 and onto, we have [26]:

$$\mathcal{P}(\mathsf{Z}|\mathsf{X}) = \mathcal{N}\left(\mathsf{Z} \mid \frac{\sum_{j,k} \sigma_j^{-2} f_j(\mathsf{X}_{j,k})}{1 + \sum_j |\mathsf{X}_j| \sigma_j^{-2}}, \frac{1}{1 + \sum_j |\mathsf{X}_j| \sigma_j^{-2}} \mathbf{I}\right)$$

Next, we define the distribution $\mathcal{P}(Y|\mathsf{Z})$. Consider a stochastic CFG which assumes (1) that a leftmost derivation is carried out, and (2) the probability distribution governing the expansion of a symbol in the grammar takes into account the sequence of all expansions so far, as well as an input value Z upon which all expansions are conditioned.

This CFG consists of productions of the form $S : seq_1 \mid seq_2 \mid seq_3 ... \mid seq_n$. Each symbol such as S corresponds to a categorical random variable with sample space $\Omega(S) = \{seq_1, seq_2, ..., seq_n\}$. A trial over the symbol S randomly selects one of the RHS sequences for that symbol. If S is a terminal symbol, then $\Omega(S) = \{\epsilon\}$, where ϵ is a special value that cannot be expanded. Subsequently, when a trial over S is performed and an RHS sequence from $\Omega(S)$ is randomly selected, we will use the sans-serif S^{rhs} to denote the identity of the RHS sequence observed.

Algorithm 1: Gen$(S, A(S)\downarrow, \mathsf{SymSoFar}, \mathsf{Z})$

Input: current symbol S, inherited attributes $A(S)\downarrow$, sequence of symbols so far $\mathsf{SymSoFar}$, latent encoding Z

Modifies: all symbols expanded are appended to $\mathsf{SymSoFar}$

Returns: $A(S)\uparrow$, the synthesized attrs of S

```
if S is a terminal symbol then
    Append (S, ε) to SymSoFar
    return ∅
else
    Choose a right-hand-side (RHS) sequence
     S^rhs ~ P(S|SymSoFar, A(S)↓, Z)
    Append (S, S^rhs) to SymSoFar
    SynthSoFar ← ⟨⟩
    for S' ∈ S^rhs in left-to-right order do
        Compute A(S')↓ from A(S)↓ and
         SynthSoFar
        A(S')↑ ←
         Gen(S', A(S')↓, SymSoFar, Z)
        Append A(S')↑ to SynthSoFar
    end
end
Compute A(S)↑ from A(S)↓ and SynthSoFar
return A(S)↑
```

Now consider a depth-first, left-to-right algorithm for non-deterministically expanding rules in the grammar to generate a *program* $\mathsf{Y} = \langle (S_1, \mathsf{S}_1^{\text{rhs}}), (S_2, \mathsf{S}_2^{\text{rhs}}), ... \rangle$; here, each S_i is a symbol encountered during the expansion, and each $\mathsf{S}_i^{\text{rhs}}$ is the identity of the RHS chosen for that symbol. Let S_1 correspond to the symbol Start. We perform a trial over S_1 and select one of the RHS sequences from $\Omega(S_1)$. Let the identity of the RHS sequence selected be $\mathsf{S}_1^{\text{rhs}}$. Note that $\mathsf{S}_1^{\text{rhs}}$ is *itself* a sequence of symbols. Choose the first symbol in the sequence $\mathsf{S}_1^{\text{rhs}}$; call this symbol S_2. Perform a trial over S_2, and let the identity of the RHS sequence chosen be $\mathsf{S}_2^{\text{rhs}}$. Choose the first symbol in $\mathsf{S}_2^{\text{rhs}}$ (call it S_3) and expand it the same way. This recursive descent continues until a terminal symbol S_i is encountered, and the recursion unwinds. If the recursion unwinds to symbol S_2, for example, then we choose the second symbol in the sequence $\mathsf{S}_1^{\text{rhs}}$, which we call S_{i+1}. We perform a trial over S_{i+1}, and let the identity of the RHS sequence chosen be $\mathsf{S}_{i+1}^{\text{rhs}}$. This sequence is recursively expanded. Once all of the symbols in the RHS associated with the Start symbol S_1 have been fully expanded, we have a program.

This generative process defines a probability distribution $\mathcal{P}(Y|\mathsf{Z})$, where for a particular program Y, the probability of observing Y is computed as

$$\mathcal{P}(\mathsf{Y}|\mathsf{Z}) = \prod_i \mathcal{P}(S_i = \mathsf{S}_i^{\text{rhs}} | S_1 = \mathsf{S}_1^{\text{rhs}}, ..., S_{i-1} = \mathsf{S}_{i-1}^{\text{rhs}}, \mathsf{Z}). \tag{1}$$

We henceforth abbreviate the expression for the inner probability as $\mathcal{P}(\mathsf{S}_i^{\text{rhs}} | \mathsf{S}_1^{\text{rhs}}, ..., \mathsf{S}_{i-1}^{\text{rhs}}, \mathsf{Z})$.

Weak Supervision with Attributes. Now assume that the grammar is an AG, so that each symbol S has an attribute set $A(S)$. We use $A(S)\uparrow$ to denote the synthesized attributes of S, and $A(S)\downarrow$ to denote the inherited attributes of S.

An NSG extends the model so that the conditional distribution $\mathcal{P}(Y|\mathsf{Z})$ is defined as:

$$\mathcal{P}(\mathsf{Y}|\mathsf{Z}) = \prod_i \mathcal{P}(\mathsf{S}_i^{\text{rhs}} | \langle \mathsf{S}_1^{\text{rhs}}, \mathsf{S}_2^{\text{rhs}}, ..., \mathsf{S}_{i-1}^{\text{rhs}} \rangle, A(S_i)\downarrow, \mathsf{Z}).$$

That is, when a symbol S_i is non-deterministically expanded, its value depends not just on the latent position Z and the sequence of expansions thus far, but also on the values of S_i's inherited attributes, $A(S_i)\downarrow$. In theory, a powerful enough learner with enough data could learn the importance of these sets of attribute values, without ever seeing them explicitly. In that sense, they could be treated as latent variables to be learned. However, the benefit of having a static analysis produce these values

deterministically is that the author of a static analysis knows the semantic rules that must be followed by a program; by presenting the data used to check whether those rules are followed directly to a learner, the process of learning to generate programs is made much easier.

Generation of a program under an NSG is described in Algorithm 1, where the distribution governing the expansion of symbol S has access to attribute values $A(S)\downarrow$.

Designing an appropriate static analysis. Intuitively, a program generated with the supervision of a static analyzer is likely to generate a semantically correct program because the static analysis provides key semantic clues during program generation. In a conventional AG-based analyzer, the AG would be used to maintain data structures that can be used to *validate* that in a complete program, key relationships hold among the values of the production's attributes. Our goal is to generate programs, rather than validate them; also, we want to *guide* the learner rather than impose hard constraints. However, constraints are a good mental model for designing a good NSG. That is, we generally expect the attribute equations used at important decision points during a neural generation process to be also helpful for validating key semantic properties of complete programs.

Example. Now we show how to use the attribute grammar in Fig. 2 in generating the body of the `write` method from Sec. 2. Let us assume that the grammar has a start nonterminal *Start* (not shown in Fig. 2) that appears in a single rule expanding it to the statement nonterminal *Stmt*. We start by extracting the context X around the method, then use this information to sample Z from $P(Z|\mathsf{X})$. Next, a Java compiler processes the surrounding code and the method's formal parameters to form the attributes $A(\textit{Start})\downarrow$, which we assume to consist of a symbol table $\{\ \texttt{f} \mapsto \texttt{File}, \texttt{str} \mapsto \texttt{String}\ \}$.

To generate a program, we sample from the distribution $P(\textit{Start}|\langle\rangle, A(\textit{Start})\downarrow, \mathsf{Z})$. First, *Start* is expanded to "*Stmt* **;** *Stmt*". When expanding the first *Stmt*, the NSG needs to choose between a method invocation and a variable declaration. Because the NSG is "aware" that this step is to expand the first line of the method—the list of RHS values chosen so far is empty—we would expect it to declare a variable. This choice gives us the RHS "*DeclType Var* **=** **new** *NewType* **(***ArgList***)**". Expanding *DeclType*, the NSG samples a Java type from the distribution

$$P(\textit{DeclType}|\langle\text{“}\textit{Stmt}\textbf{;}\ \textit{Stmt}\text{”}, \text{“}\textit{DeclType Var}\ \textbf{=}\ \textbf{new}\ \textit{NewType}\ \textbf{(}\textit{ArgList}\textbf{)}\text{”}\rangle, A(\textit{DeclType})\downarrow, \mathsf{Z}).$$

From the rules for expanding the nonterminal *DeclType* in Fig. 2, we see that the NSG can choose any Java type as the declared type of the variable. At this point, the NSG is aware that the goal is to create a method called `write` (this is encoded in Z) and that it is choosing a type to be declared on the first line of the method. It also has access to the symbol table that is maintained as part of $A(\textit{DeclType})\downarrow$. Thus, the NSG may decide to expand the symbol *DeclType* to `FileWriter`. This type is then passed upward via the synthesized attribute *DeclType*.type$\uparrow$.

Next, the grammar must expand the *Var* rule and pick a variable name to declare. This choice is returned via the synthesized attribute *Var*.name$\uparrow$. Now it is time to expand *NewType*. The attributes make this easy: when sampling from $P(\textit{NewType}|...)$, the NSG has access to *NewType*.type$\downarrow$, which takes the value `FileWriter`. A synthesizer may err by choosing a type that is not compatible with `FileWriter`. However, we may expect that during training, every time that *NewType* was expanded and the declared type was `FileWriter`, the type chosen was either `FileWriter` or some subclass of `FileWriter`. Hence the NSG is unlikely to make an error.

Assume that the NSG chooses `FileWriter`. It must now expand *ArgList*. Again, the NSG has the advantage of having access to *ArgList*.typeList$\downarrow$ (an explicit representation of the types required by the constructor being called) and, most importantly, *ArgList*.symTab$\downarrow$ (an explicit list of the variables in scope, as well as their types). At this point, it is easy for the NSG to match the required type of the first argument to the constructor (`File`) with an appropriate variable in the symbol table (`f`).

Now that the declaration of `var_0` has been fully expanded, the NSG updates the symbol table with a binding for the newly-declared variable `var_0`, and the attribute *Stmt*.symTab$\uparrow$ takes the value $\{\texttt{f} \mapsto \texttt{File}, \texttt{str} \mapsto \texttt{String}, \texttt{var\_0} \mapsto \texttt{FileWriter}\}$. When the second occurrence of *Stmt* is expanded, the symbol table is passed down via the inherited attribute *Stmt*\$1.symTab $\downarrow$. All of the information available—the latent variable Z encoding the contextual information (including the name of the method "`write`" being generated), and the symbol table containing a `FileWriter` and a `String`)—helps the NSG to deduce correctly that this *Stmt* symbol should be expanded into an invocation of a `write` method. Also, the presence of the symbol table makes it easy for the NSG to correctly attach the `write` method call to the variable `var_0` and to use `str` as the argument.

5 Evaluation

Our experimental hypothesis is that neural networks find it difficult to learn the intricate rules that govern the generation of code by only looking at the syntax of example programs. These issues become especially visible when the units of code to be generated are large, for example, entire method bodies. In contrast, an NSG can use its static analysis to compute long-distance dependencies between program variables and statements "for free." Because of this extra power, NSGs can outperform much larger neural models at generating accurate and semantically correct code.

5.1 Experimental Setup

Data. To test our hypothesis, we used a curated, deduplicated set of Java source-code files [26]. For each class and each method, we used the remainder of the class as evidence or context, and the method body was used to produce training or test data. We used 1.57 M method bodies for training. The grammar used had ten terminals corresponding to formal parameters, ten for class variables, and ten for methods local to the class. None of the Java classes in the corpus needed more than ten of each of these terminals; when generating training data, each declared Java variable or method was randomly mapped to one of the appropriate terminals. Approximately 8,000 types and 27,000 method calls from the Java JDK also appeared as terminals in the grammar.

NSG Implementation. We implemented an NSG for our subset of Java. Here, attributes are used to keep track of the state of the symbol table, the expected return type of each method, expected types of actual parameters, variable initialization, whether the variable has been used, and whether the method has a return statement. The symbol table contains entries for all formal parameters, class variables, and internal methods within the class.

The neural part of our model has 63 M parameters. To expose the attributes to the neural part of the model, we implement a depth-first search over a program's abstract syntax tree (AST) to extract node information. The attributes are then encoded in a standard way — for example, the symbol table is represented as matrix (rows correspond to types, columns to variables, the value 1 is present if the corresponding type/variable pair is in scope). The distribution $\mathcal{P}(\mathsf{S}_i^{\text{rhs}} | \langle \mathsf{S}_1^{\text{rhs}}, \mathsf{S}_2^{\text{rhs}}, ..., \mathsf{S}_{i-1}^{\text{rhs}} \rangle, A(S_i)\downarrow, \mathsf{Z})$ is implemented as a set of LSTMs that decode the sequence of symbols, as well as the encoded $A(S_i)\downarrow$ and Z, into a distribution over $\mathsf{S}_i^{\text{rhs}}$. We trained our framework on top of Tensorflow [1]. Using one GPU, the NSG training time is around 72 hours. See Appendix C for more details.[1]

Baselines. We consider three categories of baselines. The first consists of large pretrained transformers. Specifically, we consider two variants of the GPT-NEO [6] model with 125 M and 1.3 B parameters. Both models are pre-trained on the Pile dataset [15], which consists of an 800 GB English-text corpus and open-source code repositories. On the APPS dataset [17], they perform well compared to OpenAI's 12-B-parameter, GPT-3-like CODEX model [8]. We also compare against CODEGPT [24] which is a GPT-2-like model with 125 million parameters. This model was pretrained on Python and Java corpora from the CodeSearchNet dataset, which consists of 1.1 M Python functions and 1.6 M Java methods. We fine-tune all of these pretrained models on our Java dataset, using the token-level code-completion task provided by CodeXGLUE [24]. Finally, we also offer a comparison against the CODEX model [8]. Because we did not have access to the model's pretrained weights, this model is only used in a zero-shot fashion (no fine-tuning on our Java dataset). It should be noted here that the transformer baselines work on the entire Java language, whereas our NSG framework works on a sub-part of Java which is supported in our grammar definition.

The second category comprises an ablation, called a "conditional neural grammar" (CNG), that is identical to our NSG model but is trained without any of the attribute information. In other words, the CNG model is trained only on the program syntax. The third category includes GNN2NAG [7], a graph-neural-network-based method that uses an attribute grammar but *learns* the attributes from data rather than computing them symbolically. See Appendix C for more details on the baselines.

Test Scenario. Our test scenario is as follows. Given a Java class, we remove the entire body of a randomly selected method. We then use the remaining potion of the class along with the method header as context information that is then fed to the model as input. We run our NSG model and the baselines to regenerate this method body conditioned on the resulting context. We report the accuracy of the prediction based on static-semantic checks and fidelity measures.

[1] Our implementation is available at https://github.com/rohanmukh/nsg.

Table 1: Percent of Static Checks Passed

	GPTNeo125M	GPTNeo1.3B	CODEX	CODEGPT	GNN2NAG	CNG	NSG
No undeclared variable access	89.87%	90.36%	88.62%	90.94%	47.44%	19.78%	**99.82%**
Valid formal parameter access	NA	NA	NA	NA	25.78%	11.03%	**99.55%**
Valid class variable access	NA	NA	NA	NA	15.40%	12.75%	**99.53%**
No uninitialized objects	93.90%	91.73%	90.82%	94.37%	21.20%	21.56%	**99.01%**
No variable access error	90.36%	90.51%	88.86%	91.32%	28.92%	17.92%	**99.69%**
Object-method compatibility	**98.36%**	98.09%	98.35%	97.84%	21.43%	12.23%	97.53%
Return type at call site	97.38%	98.01%	**98.53%**	97.83%	23.86%	16.40%	98.01%
Actual parameter type	87.03%	86.36%	92.28%	88.71%	9.27%	16.09%	**97.96%**
Return statement type	84.05%	85.09%	88.13%	85.23%	12.34%	9.51%	**90.97%**
No type errors	87.25%	88.13%	91.42%	88.10%	16.31%	13.56%	**97.08%**
Return statement exists	99.61%	99.80%	98.44%	99.57%	94.02%	**99.92%**	97.10%
No unused variables	96.42%	96.46%	96.82%	**97.64%**	20.95%	24.29%	93.84%
Percentage of parsing	98.18%	98.13%	96.41%	97.08%	**100.0%**	**100.0%**	**100.0%**
Pass all checks	65.26%	64.88%	47.49%	67.73%	17.34%	12.87%	**86.41%**

Table 2: Average Fidelity of Generated Method Bodies

	GPTNeo125M	GPTNeo1.3B	CODEX	CODEGPT	GNN2NAG	CNG	NSG
Set of API Calls	32%	37%	36%	36%	3%	22%	**53%**
Sequences of API Calls	17%	20%	16%	19%	0.3%	18%	**42%**
Sequences of Program Paths	12%	15%	10%	14%	0%	17%	**39%**
AST Exact Match	12%	15%	10%	14%	0%	6%	**26%**

5.2 Results

Static Checks. For each generated method body, we check the following properties: *(1) No undeclared variable access:* Are all the variables used in a program declared (within an enclosing scope) before they are used? *(2) Valid formal parameter access:* Are formal parameters that are used in the method body present in the method declaration? *(3) Valid class-variable access:* Are the class variables that are used in the method body present in the class declaration? *(4) No uninitialized objects:* Do variables have a non-null value when they are used? *(5) No variable access errors:* Are checks (1)-(4) all satisfied? *(6) Object-method compatibility:* Are methods called on objects of a given class actually available within that class? *(7) Return type at the call site:* Is the assignment of the return value type-correct with respect to the return type of the called method? *(8) Actual-parameter type:* Are the actual-parameter types in an API call consistent with the corresponding formal-parameter types? *(9) Return-statement type:* Is the type of the expression in a return statement consistent with the method's declared return type? *(10) No type errors:* Are checks (6)-(10) all satisfied? *(11) Return statement exists:* Does the method body have a return statement somewhere? *(12) No unused variables:* Are all variables declared in the method body used in the method? *(13) Percentage of parsing:* Can the generated method be parsed by a standard Java parser? *(14) Pass all checks:* Are checks (1)-(13) all satisfied?

Note that (2) and (3) are not meaningful metrics for approaches, such as our transformer baselines, that do not use a grammar to generate code. This is because in these models, when a variable token is generated, there is no way to tell what category of variable (class variable, formal parameter, etc.) it is meant to be. These metrics are meaningful for the NSG, CNG, and GNN2NAG models, which use a Java parser capable of partitioning variable names into different categories.

The results of our comparisons appear in Table 1. These scores are interpreted as follows. Suppose that a generated program uses five variables, of which four are declared correctly in the proper scope. This situation is scored as 80% correct on the "No undeclared-variable access" criterion. We report the average success rate over each of these properties over all the generated programs in our test suite.

Whole-Method Fidelity. We also check the *fidelity* of the generated code to the reference code. One possibility here is to use a standard metric for text generation, such as the BLEU score. However, this is problematic. As the BLEU score is not invariant to variable renamings, a nonsensical program that uses commonplace variable names can get an artificially high BLEU score. Also, programs are structured objects in which some tokens indicate control flow and some indicate data flow. The BLEU score does not take this structure into account. See Appendix J for a concrete example of these issues.

Query	`public class FileUtils{`
	` FileReader field_7;`
	` BufferedReader field_5;`
	
	` /** read line from file */`
	` public String reader () {}}`

NSG	`public String reader(){`
	` java.lang.String var_9;`
	` try {var_9=field_5.readLine();`
	` } catch (IOException var_8) {`
	` var_8.printStackTrace(); }`
	` return var_9; }`

CodeGPT	`public String reader(){`
	`  StringBuffer buffer=`
	`  new StringBuffer();`
	`  buffer.append("\n");`
	`  return buffer.toString();`
	` }`

CODEX	`public String reader()`
	` throws IOException{`
	` field_5= new BufferedReader(`
	` new FileReader(field_7));`
	` return field_5.readLine(); }`

GptNeo1.3B	`public String reader()`
	` throws IOException {`
	` try {`
	` field_7=new FileReader(this.file);`
	` field_5=new BufferedReader(field_7);`
	` String line;`
	` while (field_5.readLine()) {`
	` System.out.println(line); }`
	` return line;`
	` } catch (FileNotFoundException e)`
	` {e.printStackTrace(); }`
	` return null; }`

Table 3: Reading from a file: Outputs for the NSG and transformer baselines.

Instead, we consider four fidelity metrics: (1) *Set of API Calls*: Extract the set of API calls from the generated and reference codes, and compute the Jaccard similarity between the sets. (2) *Sequences of API Calls*: Generate the set of all possible API call sequences possible along code paths, and compute the Jaccard similarity between the sets for the generated and reference code. (3) *Sequences of Program Paths*: Generate the set of all possible paths from root to leaf in the AST, then compute the Jaccard similarity between the sets (two paths are equal if all elements except for object references match). (4) *AST Exact Match*: Exact AST match (except for object references), scored as 0 or 1. We compute the highest value for each metric across the ten bodies generated, and average the highest across all test programs. Results for these measures are presented in Table 2.

Summary of results. We find that in most cases, the NSG had a higher incidence of passing the various static checks compared to the baselines. This is perhaps not surprising, given that the NSG has access to the result of the static analysis via the attribute grammar. More intriguing is the much higher accuracy of the NSG for the fidelity results. Pre-trained language models and GNN2NAG are designed for next-token-prediction tasks (we give some results on these tasks in Appendix G). However, in our CPG task, no tokens are available from the method body to be generated. In particular, language models must treat the surrounding code and method header as input from which to generate the entire method body, and this proves difficult. The NSG, on the other hand, uses static analysis to symbolically extract this context, which is explicitly given to the neural network in the form of the class variables and methods that are available to be called (in $A(S)\downarrow$), and in the class name, encoded comments, variable names, and so on (in Z).

Transformers vs. NSGs: As a complement to our quantitative evaluation, we manually examined the outputs of our model and the baselines on a set of hand-written tasks for qualitative evaluation. The transformers produced impressively human-like code on several of these tasks. However, in quite a few cases, they produced incorrect programs that a trained human programmer would be unlikely to write. Also, the transformers were biased towards producing short programs, which often led them to produce uninteresting outputs.

Table 3 illustrates some of the failure modes of the transformer baselines. Here, we consider the task of reading a string from a file utility class. The top result for our NSG model declares a *String* variable to read from the already existing field while also correctly catching an *IOException*. The CODEGPT output in this case is unrelated to the context. CODEX initiates a FileReader object by invoking an argument which is of type *FileReader* itself, thereby causing a type mismatch. The code from GPT-NEO accesses a *file* instance variable that does not exist and also returns a blank line from the method. A few other examples of NSG and transformer outputs appear in Appendix A.

6 Related Work

Non-Neural Models of Code. Many non-neural models of code have been proposed over the years [25, 32, 28, 2, 27, 5]. A few of these models condition generation on symbolic information from the context. Specifically, Bielik et al. [5] use programmatically represented functions to gather

information about the context in which productions for program generation are fired, then utilize this information to impose a distribution on rules. Maddison & Tarlow [25] generate programs using a model that encodes a production's context using a set of "traversal variables." However, the absence of neural representations in these models puts a ceiling on their performance.

Deep Models of Code. There is, by now, a substantial literature on deep models trained on program syntax. Early work on this topic represented programs as sequences [32] or trees [26, 38, 9], and learned using classic neural models, such as RNNs, as well as specialized architectures [23, 30, 3]. The recent trend is to use transformers [36, 16, 14, 24]. Some of these models — for example, CODEGPT [24] — are trained purely on code corpora (spanning a variety of languages, including Java). Other models, such as CODEBERT [14], GPT-NEO [6], and CODEX [8], are trained on both natural language and code. In all of these cases, programs are generated without any explicit knowledge of program syntax or semantics.

The GNN2NAG model by Brockschmidt et al. [7] also uses an attribute grammar to direct the generation of programs. However, unlike our method, this model use a graph neural net to *learn* attributes of code. Our experiments show the benefits of our weak supervision approach over this.

Also related is work by Dai et al. [11], who extend grammar variational autoencoders [21] with hard constraints represented as attribute grammars. In that work, attribute constraints are propagated top-down, and every generated artifact is required to satisfy the top-level constraint. This strategy comes with challenges; as is well-known in the program-synthesis literature [31], top-down constraint propagation can lead to unsatisfiability, and require rejection of generated samples, for grammars above a certain level of complexity. We sidestep this issue by using attribute grammars as a form of weak supervision, rather than as a means to enforce hard constraints.

Neurally Directed Program Synthesis. Many recent papers study the problem of *neurally directed program synthesis* [4, 12, 34, 10, 33, 29]. Here, neural networks, and sometimes program analysis, are used to guide a combinatorial search over programs. Because such search is expensive, these methods are typically limited to constrained domain-specific languages. In contrast, our approach does not aim for a complete search over programs at generation time (our decoder does perform a beam search, but the width of this beam is limited). Instead, we embody our program generator as a neural network that sees program-analysis-derived facts as part of its data. This design choice makes our method more scalable and allows it to handle generation in a general-purpose language.

7 Conclusion

We have presented a framework for deep generation of source code in which the training procedure is weakly supervised by a static analyzer, in particular, an attribute grammar. We have shown that our implementation of this approach outperforms several larger, state-of-the-art transformers both in semantic properties and fidelity of generated method bodies.

A lesson of this work is that while modern transformers excel at writing superficially human-like code, they still lack the ability to learn the intricate semantics of general-purpose languages. At the same time, the semantics of code can be defined rigorously and partially extracted "for free" using program analysis. This extracted semantics can be used to aid neural models with the concepts that they struggle with during program generation. While we have used this idea to extend a tree LSTM, we could have implemented it on top of a modern transformer as well. We hope that future work will pursue such implementations.

Our work demonstrates an alternative use for formal language semantics, compared to how semantics are typically used in program synthesis research. Historically, semantics have been used to direct generation-time combinatorial searches over programs. However, scalability has been a challenge with such approaches. Our work points to an alternative use of semantics: rather than using semantic program analyses to direct a search over programs, one could use them to *annotate* programs at training and test time and leave the search to a modern neural network. We believe that such a strategy has the potential to vastly extend the capabilities of algorithms for program synthesis.

Acknowledgments. This work was funded by NSF grants 1918651 and 2033851; US Air Force and DARPA contract FA8750-20-C-0002; and ONR grants N00014-17-1-2889, N00014-19-1-2318, and N00014-20-1-2115. We thank Matt Amodio for his help with our initial exploration of the problem studied here, and Martin Rinard and the anonymous referees for their valuable comments.

References

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Allamanis, M. and Sutton, C. Mining idioms from source code. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pp. 472–483, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3056-5. doi: 10.1145/2635868.2635901.

[3] Alon, U., Zilberstein, M., Levy, O., and Yahav, E. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[4] Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., and Tarlow, D. Deepcoder: Learning to write programs. *International Conference on Learning Representations (ICLR)*, 2017.

[5] Bielik, P., Raychev, V., and Vechev, M. PHOG: Probabilistic model for code. In *ICML*, pp. 19–24, 2016.

[6] Black, S., Gao, L., Wang, P., Leahy, C., and Biderman, S. GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow, March 2021.

[7] Brockschmidt, M., Allamanis, M., Gaunt, A. L., and Polozov, O. Generative code modeling with graphs. In *International Conference on Learning Representations*, 2018.

[8] Chen, M., Tworek, J., Jun, H., Yuan, Q., Ponde, H., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[9] Chen, X., Liu, C., and Song, D. Tree-to-tree neural networks for program translation. *Advances in Neural Information Processing Systems*, 31, 2018.

[10] Chen, Y., Wang, C., Bastani, O., Dillig, I., and Feng, Y. Program synthesis using deduction-guided reinforcement learning. In *International Conference on Computer Aided Verification*, pp. 587–610. Springer, 2020.

[11] Dai, H., Tian, Y., Dai, B., Skiena, S., and Song, L. Syntax-directed variational autoencoder for structured data. In *International Conference on Learning Representations*, 2018.

[12] Devlin, J., Uesato, J., Bhupatiraju, S., Singh, R., Mohamed, A.-r., and Kohli, P. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*, pp. 990–998. PMLR, 2017.

[13] Devlin, J., Chang, M., Lee, K., and Toutanova, K. BERT: pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT (1)*, 2019.

[14] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al. Codebert: A pre-trained model for programming and natural languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings*, pp. 1536–1547, 2020.

[15] Gao, L., Biderman, S. R., Black, S., Golding, L., Hoppe, T., Foster, C., Phang, J., He, H., Thite, A., Nabeshima, N., Presser, S., and Leahy, C. The pile: An 800gb dataset of diverse text for language modeling. *ArXiv*, abs/2101.00027, 2021.

[16] Gemmell, C., Rossetto, F., and Dalton, J. Relevance transformer: Generating concise code snippets with relevance feedback. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 2005–2008, 2020.

[17] Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D., and Steinhardt, J. Measuring coding challenge competence with apps. *NeurIPS*, 2021.

[18] Hopcroft, J. E., Motwani, R., and Ullman, J. D. Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):60–65, 2001.

[19] Javalang. Pure Python Java parser and tools, March 2020. https://pypi.org/project/javalang/.

[20] Knuth, D. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968.

[21] Kusner, M. J., Paige, B., and Hernández-Lobato, J. M. Grammar variational autoencoder. In *International Conference on Machine Learning*, pp. 1945–1954. PMLR, 2017.

[22] Li, Y., Tarlow, D., Brockschmidt, M., and Zemel, R. Gated graph sequence neural networks. *CoRR*, abs/1511.05493, 2016.

[23] Ling, W., Blunsom, P., Grefenstette, E., Hermann, K. M., Kočiskỳ, T., Wang, F., and Senior, A. Latent predictor networks for code generation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, (Volume 1)*, pp. 599–609, 2016.

[24] Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, D., Jiang, D., Tang, D., et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

[25] Maddison, C. and Tarlow, D. Structured generative models of natural source code. In *ICML*, pp. II–649–II–657, 2014.

[26] Murali, V., Qi, L., Chaudhuri, S., and Jermaine, C. Neural sketch learning for conditional program generation. In *ICLR*, 2018.

[27] Nguyen, A. T. and Nguyen, T. N. Graph-based statistical language model for code. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pp. 858–868, Piscataway, NJ, USA, 2015. IEEE Press. ISBN 978-1-4799-1934-5.

[28] Nguyen, T. T., Nguyen, A. T., Nguyen, H. A., and Nguyen, T. N. A statistical semantic language model for source code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pp. 532–542, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2237-9. doi: 10.1145/2491411.2491458.

[29] Odena, A. and Sutton, C. Learning to represent programs with property signatures. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[30] Parisotto, E., Mohamed, A.-r., Singh, R., Li, L., Zhou, D., and Kohli, P. Neuro-symbolic program synthesis. In *ICLR*, 2017.

[31] Polikarpova, N., Kuraj, I., and Solar-Lezama, A. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices*, 51(6):522–538, 2016.

[32] Raychev, V., Vechev, M., and Yahav, E. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 419–428, 2014.

[33] Shah, A., Zhan, E., Sun, J. J., Verma, A., Yue, Y., and Chaudhuri, S. Learning differentiable programs with admissible neural heuristics. In *Advances in Neural Information Processing Systems*, 2020.

[34] Si, X., Yang, Y., Dai, H., Naik, M., and Song, L. Learning a meta-solver for syntax-guided program synthesis. In *International Conference on Learning Representations*, 2019.

[35] Sutskever, I., Vinyals, O., and Le, Q. V. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pp. 3104–3112, 2014.

[36] Svyatkovskiy, A., Deng, S. K., Fu, S., and Sundaresan, N. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1433–1443, 2020.

[37] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. In *Advances in neural information processing systems*, pp. 5998–6008, 2017.

[38] Yin, P. and Neubig, G. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, pp. 440–450, 2017.

	(a) Removing from a list	(b) Adding to a list

Query	`public class myClass{`
	
	`  /** remove items from a list */`
	`  public void`
	`  remove(List<String> fp_2){`
	`  }`
	`}`
		`public class myClass{`
		
		`  /** add item to list */`
		`  public void addItem (`
		`  List<String> a, String b){`
		`  }}`

NSG	`void remove(java.util.List`
	`  <java.lang.Object> fp_2){`
	`  java.util.Iterator var_8;`
	`  var_8 = fp_2.iterator();`
	`  while (var_8.hasNext()){`
	`  java.lang.Object var_3;`
	`  var_3 = new java.lang.Object(`
	`  (java.lang.String) ARG,`
	`  (javax.swing.Icon) ARG);`
	`  var_3 = var_8.next();`
	`  var_8.remove();`
	`}`
	`return;`
	`}`
		`public void addItem (`
		`  List<String> fp_9,`
		`  String fp_1) {`
		`  if (fp_9.contains`
		`  ((String) ARG)) {`
		`  fp_9.add(fp_1);`
		`  }`
		`  return;`
		`}`

CODEGPT	`public void remove(List<String> fp_2){`
	`  items.remove(fp_2); }`
		`public void addItem (`
		`  List<String> a, String b)`
		`{`
		`  items.add(a);`
		`}`

CODEX	`public void remove(List<String> fp_2){`
	`  fp_2.remove(0); }`
		`public void addItem (`
		`List<String> a, String b) {`
		`a.add(b); }`

GptNeo1.3B	`public void remove(List<String> fp_2){`
	`  fp_2.remove(); }`
		`public void addItem (`
		`List<String> a, String b) {`
		`  List <String> temp =`
		`  new ArrayList<String>();`
		`  for (int i=0; i<a.size(); i++){`
		`  temp.add(a.get(i));`
		`  }`
		`a.add(b);`
		`}`

Table 4: Example synthesis outputs: (a) Removing from a list and (b) Adding to a list.

A Additional Synthesis Examples

Some additional program-generation examples are shown in Tables 4, 5, and 6.

B Static Checks Considered

Here we give an in-depth description of each of the static checks that has been tested in the paper as described in Section 5.2.

- **No Undeclared Variable Access.** All the variables used in a program should be declared before they are used and should be available in the scope. We measure the percentage of variable usages across all the programs that are declared before use. For example `int bar() {x.write();}` is a violation of this property because `x` is not declared (assuming that there is no field named `x`). When the statement `x.write()` is synthesized, the NSG-model has access to the symbol table at that point in the AST. The symbol table does not contain the variable `x` because it is not declared in the preceding statements. The NSG-model has learnt to use variables that are present in the symbol table (encoded as attribute "symTab"), which increases the likelihood that the variables used in the program are declared before being used.

- **Valid Formal Parameter Access.** All of the input variables accessed in a program body should be available in the class definition. Across all programs, we measure the percentage of input-variable accesses that are available. If a program grammar allows access for n input variables, and the synthesizer tries to access one such variable n_k even when it is not

	(a) Writing to a file	(b) Creating a Swing JButton

Query	`public class FileUtil{`
	
	`  String err;`
	`  public int read(File f){...}`
	
	`  /* write lines to file */`
	`  public void write(`
	`  File f, String str){}}`
		`public class myGUI{`
		
		`  public javax.swing.JFrame`
		`  createFrame(final String title)`
		`  {... }`
		
		`  public void addActionListener`
		`  (javax.swing.JButton title)`
		`  {... }`
		
		`  /**`
		`  create a button`
		`  */`
		`  public`
		`  javax.swing.JButton createButton(){ }`
		`}`

NSG	`void write(File f, String str){`
	`  try {`
	`  FileWriter var_0;`
	`  var_0 = new FileWriter(f);`
	`  var_0.write(str);`
	`  } catch(IOException var_0) {`
	`  var_0.printStackTrace();`
	`  System.out.println( ARG ); }`
	`  return; }`
		`javax.swing.JButton createButton(){`
		`  javax.swing.ImageIcon var_9;`
		`  var_9 = new javax.swing.ImageIcon.`
		`  ImageIcon(`
		`  (java.lang.String) ARG,`
		`  (java.lang.String) ARG`
		`  );`
		`  javax.swing.JButton var_7;`
		`  var_7 = new javax.swing.JButton.JButton(`
		`  java.lang.String: ARG,`
		`  javax.swing.Icon: var_9);`
		`  var_7.setVerticalTextPosition(`
		`  (int) ARG);`
		`  var_7.setHorizontalTextPosition(`
		`  (int) ARG);`
		`  return var_7;`
		`}`

CODEGPT	`void write (File a, String b){`
	`System.out.println(a);`
	`}`
		`public JButton createButton(){`
		`  return new JButton(this);`
		`}`

CodeX	`public void write(File f, String str){`
	`  Writer writer = null;`
	`  try {`
	`  writer = new BufferedWriter(`
	`  new FileWriter(f));`
	`  writer.write(str);`
	`  writer.close();`
	`  } catch (IOException e) {`
	`  err = e.printStackTrace();}`
	`}`
		`public JButton createButton(){`
		`  final javax.swing.JButton b =`
		`  new javax.swing.JButton();`
		`  return b;`
		`}`

GptNeo1.3B	`public void write(File f, String str)`
	`  { try { f.write(str); }`
	`  catch ( IOException e) {`
	`  e.printStackTrace(); }`
	`}`
		`public JButton createButton(){`
		`  javax.swing.JButton button =`
		`  new javax.swing.JButton();`
		`  button.addActionListener(this);`
		`  return button;`
		`}`

Table 5: Example synthesis outputs: (a) Writing to a file and (b) Creating a Swing JButton. For brevity, we omit the method body in the query, denoted as **...**. The method body was fed into transformers as a part of the prompt during generation.

	(c) Connect to a Socket	(d) Decrypting a Message

Query	`public class tcpSocketManager{`
	
	`  public Socket createSocket(final InetSocketAddress socketAddress){`
	`  ...`
	`  }`
	
	`  public void connect( InetSocketAddress socketAddress){`
	`  }`
	`}`
		`public class lockAndKey{`
		
		`  public String encryptMessage( String message, String secret_key){`
		`  ...`
		`  }`
		
		`  public String decryptMessage( String encrypted_message, String secret_key){`
		`  }`
		`}`

NSG	`public void connect( InetSocketAddress fp_3)`
	`  Socket var_8;`
	`  var_8 = new Socket.Socket( (String) ARG,(int) ARG);`
	`  try {`
	`  InetSocketAddress var_3;`
	`  var_3 = new InetSocketAddress. InetSocketAddress((String) ARG, (int) ARG);`
	`  var_8.connect((SocketAddress) var_3, (int) ARG);`
	`  }`
	`  catch(ConnectException var_3){`
	`  var_3.printStackTrace();`
	`  }`
	`  try {`
	`  var_8.close();`
	`  }`
	`  catch(IOException var_3){`
	`  var_3.printStackTrace();`
	`  }`
	`  return;`
		`String decryptMessage(String fp_2, String fp_5){`
		`  if (fp_5.equals((Object) ARG)){`
		`  return fp_5;`
		`  }`
		`  int var_6;`
		`  var_6 = fp_5.indexOf((String) fp_5);`
		`  if (true){`
		`  return fp_5;`
		`  }`
		`  String var_4;`
		`  var_4 = fp_5.substring((int) ARG, (int) var_6);`
		`  return var_4;`
		`}`

CODEGPT	`public void connect(InetSocketAddress socketAddress)`
	`  if (socketAddress ~= null) {`
	`  socketAddress.connect(socketAddress);`
	`}`
		`public String decryptMessage( String encrypted_message, String secret_key){`
		`return decryptMessage(encrypted_message, secret_key);`
		`}`

CODEX	`public void connect(InetSocketAddress socketAddress){`
	`  AbstractSocket activeProxySocket =`
	`  createSocket(socketAddress);`
	`  activeProxySocket.connect(`
	`  socketAddress);`
	`}`
		`public String decryptMessage( String encrypted_message, String secret_key){`
		`  Key aesKey = new SecretKeySpec(`
		`  secret_key.getBytes(), 'AES');`
		`  Cipher cipher =`
		`  Cipher.getInstance('AES');`
		`  cipher.init(Cipher.DECRYPT_MODE, aesKey);`
		`  return cipher.doFinal(encrypted_message);`
		`}`

GptNeo1.3B	`public void connect(InetSocketAddress socketAddress)`
	`  throws IOException {`
	`  Socket s = new Socket(`
	`  new Proxy(`
	`  Proxy.Type.SOCKS,socketAddress)`
	`  );`
	`  s.connect();`
	`}`
		`public String decryptMessage( String encrypted_message, String secret_key){`
		`  Key aesKey=`
		`  new SecretKeySpec(`
		`  encrypted_message.getBytes(),`
		`  secret_key.getBytes()`
		`  );`
		`  return aesKey.doFinal(`
		`  encrypted_message);`
		`}`

Table 6: Example synthesis outputs: (a) Connect to a socket and (b) Decrypting a message. For brevity, we omit the method body in the query, denoted as **....** The method body was fed into transformers as a part of the prompt during generation.

available, this property will be violated. The formal-parameter-type information is present in *symTab* corresponding to each of the input variable which helps NSG learn this property correctly.

- **Valid Class Variable Access.** All the class fields accessed in a program body should be available in the class definition. The presence of field information in *symtab* helps NSG satisfy this semantic property. Across all programs, we measure the percentage of field accesses that happened when they were available.

- **No Uninitialized Objects.** All variables with reference type should be initialized. Out of all the variables with reference types declared across all the programs, we measure the percentage of variables that are initialized using a "`new`" statement. For example, `BufferedWriter x; x.write();` is a violation because `x` is not initialized using `new BufferedWriter`. Violation of this property could cause a *NullPointerException* at runtime. The AG keeps track of variable initializations using an attribute of type array of Booleans, named *IsInitialized*. Whenever a variable is declared, the corresponding value in the *IsInitialized* array is set to *False*. As soon as the variable is initialized, the attribute is set to *True*. This attribute helps the NSG model learn to avoid generating method bodies in which a variable is used without being initialized.

- **No Variable Access Errors.** This property is the aggregate of the preceding four semantic checks.

- **Object-method compatibility.** Methods should be invoked on objects of appropriate types. For example, consider the program snippet `int k = 0; bool b = k.startsWith(pre);` which contains an invocation of `String::startsWith(String pre)`. This program fails the object-method-compatibility check because the method is invoked on a variable of type *int* instead of a variable of type *String*. The method invocation *startsWith* is synthesized before the terms *k* and *pre*.[2] The symbol-table attribute of the AG, in combination with the synthesized attribute *expr_type*, helps the NSG model avoid such cases by learning to synthesize an expression with a compatible type, given the method signature to be invoked on it.

- **Return Type at the Call Site.** The return type of a method invoked at some call site should be consistent with the type expected at the call site. In `bool b = aStr.startsWith(pre);` this property asserts that the type of `b` should be compatible with the return type of `String::startsWith`. The symbol table alongside the synthesized attribute from the API call, namely *ret_type*, helps the NSG respect this rule.

- **Actual Parameter Type.** The actual-parameter types in an API call should be consistent with the corresponding formal-parameter types. For example, consider the program fragment `int pre = 0; bool b = aStr.startsWith(pre);` which contains an invocation of `String::startsWith(String pre)`. This program fails the formal-parameter type check because the actual-parameter type (*int*) does not match the formal-parameter type (*String*). The AG has the symbol-table attribute, which contains the variables in scope and their types, plus it has access to the intended type from the API call by the attribute *typeList*, which helps the NSG model to learn to synthesize API arguments of appropriate types.

- **Return Statement Type.** The type of the expression in a return statement of a method should be consistent with the method's declared return type. For example, `public int foo(){String x; return x}` violates this property because the returned expression `x` is of type *String*, whereas the declared return type is *int*. To make it easier for the NSG model to learn this restriction, the AG has a dedicated attribute *methodRetType* to propagate the declared return type throughout the method, which helps it generate an expression of the appropriate type after consulting the symbol table. For this property, we measure the

[2]Note that while this attribute grammar requires *Method* to be expanded before *Expr* (because inherited attributes of the latter depend on synthesized attributes of the former), the grammar is still *L*-attributed if we expand *Method* then *Expr*, and perform an unparsing trick to emit the subtree produced by *Expr* first.

percentage of return statements for which the expression type matches the method's declared return type.

- **No Type Errors.** All variables should be accessed in a type-consistent manner. Across all the programs, we measure the percentage of variable accesses that are type-consistent. Locations of variable accesses include all types of variable accesses relevant to an API call, method arguments, return statements, the variables on which the methods are invoked, variable assignments, and internal class method calls.

- **Return Statement Exists.** This property asserts that a method body should have a return statement. `public int foo(){String x;}` violates this property because the method body does not have a return statement. The AG propagates an attribute, *retStmtGenerated*. This attribute is initially set to *false*. When a return statement is encountered, the attribute is set to *true*. The NSG model learns to continue generating statements while this attribute is *false*, and to stop generating statements in the current scope when the attribute is *true*. For this property, we report the percentage of programs synthesized with a `return` statement.

- **No Unused Variables.** There should not be any unused variables (variables declared but not used) in the method body. For example, in `public void reader(){String x; String y; x=field1.read()}`, the variable `y` is an unused variable. To keep track of the unused variables, we use a boolean array attribute *isUsed*. Entries in array corresponding to the used variables are *true* whereas all other entries are *false*. Out of all the programs synthesized, we report the percentage of variables declared which have been used inside the method body.

- **Percentage of Parsing.** A parser for the Java language should be able to parse the synthesized programs. We use an open-source Java parser, called javalang [19], and check for the number of programs that parse. This test does not include static-semantic checks; it only checks if a generated program has legal Java syntax. Note that NSG, CNG, and GNN2NAG models are rule-based generation and they are bound to parse by definition. The pre-trained language models, however, are not guaranteed to produce programs that exactly follow the grammar definition. Therefore we capture all such instances that throw parsing exceptions and report the resulting numbers.

- **Pass All Checks.** This property is the aggregate of all of the preceding checks.

C Implementation Details

We now give a few details about how the distributions required to instantiate an NSG are implemented in our Java prototype.

Evidence Encoder: The evidences that we support as input to user context include class-level information (e.g., class name, Java-type information of the instance variables, and methods in the same class that have already been implemented); along with the information from the method header.

Each of these evidence types is encoded in a way appropriate to its domain. The method header has a separate encoding for each of its components: return type, formal parameters, and method name. The natural-language description available as Javadoc is also included. In total, there are seven kinds of evidence that we consider in our context.

The evidences are encoded together as follows: class and method names are split using camel case and delimiters; the resulting elements are treated as natural-language keywords, and encoded as a set, using a single-layer feed-forward network. The other evidences that are represented as sets and encoded by similar neural network. The type information of the class variables, formal parameters, and Javadoc are encoded as sequential data using a single-layered LSTM. The surrounding method is encoded as a concatenation of the three components of the method header, namely, the method name, formal parameters, and return type, followed by a dense layer to reduce the dimensionality to the size of the latent space. Note that the model defined in Section 4 allows us to get meaningful synthesis outputs even when only a small subset of the different kinds of evidence are available during training.

Sampling Symbol RHS Values: The distribution $P(S|\mathsf{SymSoFar}, A(S)\downarrow, \mathsf{Z})$ is implemented using an LSTM. There are six different kind of symbols for which we need to chooses an RHS: choosing the program block to produce (e.g., producing a try-catch statement or a loop), Java types, object-initialization calls, API calls, variable accesses, and accessing a method within the same class. Each one of these has their own vocabulary of possibilities, and requires a separate neural LSTM decoding unit. It is also possible to use additional, separate neural units in different production scenarios. In our implementation, we use four separate LSTM units for decoding variable accesses: for a variable that is being declared, when accessed in a return statement, when accessed as an input parameter, or when an API call is invoked. In other words, the NSG synthesizer consists of multiple decoding neural units, for decoding all of the production rules in the program's grammar, each using a separate LSTM unit. It should be noted here that even though each of these LSTM units in the network has its own parameter set, they all maintain the same recurrent state, which tracks the state of the unfinished program synthesized so far.

Attributes: Each of the neural units in an NSG decodes the current symbol using its corresponding LSTM and additional attributes available from the attribute grammar. Generally when a recurrent model like an LSTM is trained, the input to the LSTM cell is fixed as the correct output from the last time step (or the output from the parent node in case of a tree decoder). The availability of attributes in an NSG lets us augment this input information with the additional attributes from our grammar. The attributes that we support are given below:

- Symbol table: An unrolled floating-point matrix that represents the types of all variables in scope, including field variables, input variables, and user-defined variables. Represented in our grammar as *symTab* attrbute.
- Method return type: A floating-point vector containing the expected type of the method body. Represented in our grammar as *methodReturnType*.
- Return type of an API call, expression type of an object invoking an API call, and types of the input variables of an API call: Three separate floating-point vectors that represent the expected return type (*retType*), the expression type of the object that initiates an API call (*exprType*), and the expected formal parameters of the API call, if any (*typeList*).
- Internal-method table: A floating-point vector representing the neural representation of the completed methods available in the same class.
- Unused variable flag: A Boolean vector indicating which variables have been initialized but not used so far in the program. The attribute that tracks this semantic property is *isUsed*.
- Uninitiated-object flag: A Boolean vector indicating which objects have been declared but not initialized. The attribute that tracks this semantic property is *isInitialized*.
- Return-statement flag: A Boolean indicating if a return statement has yet been reached in the program body. The attribute that tracks this semantic property is *retStmtGenerated*

Note that not all attributes are important to every production rule in the grammar at a given time step. For example, while decoding a variable access, it is unimportant to know about internal methods. This information follows from our attribute grammar, as described in Appendix K. If a particular attribute is not associated with a non-terminal, or it is unused inside a production rule, it is not required required for that rule, and the attribute can be omitted from being input to decoding that particular token.

Training and Inference: During training, all information required for the LSTM, such as contextual information for the missing class and the associated attributes in the AST, are available to the neural decoder. The neural network is only tasked with learning the correct distributions related to decoding the method-body AST. The objective function related to learning the probability distributions within the learner is composed of a linear sum of cross-entropy loss for each category of symbol: non-terminals of the AST, API calls, Java types, and so on. This loss can be minimized using standard techniques like gradient descent to 'train' the model. If we compare this to a simple neural model, the NSG decoder has additional inputs in the form of attributes coming from the grammar to aid its decoding process.

During inference, only the program context is available to the decoder. The attributes that the synthesizer requires are inferred on-the-fly, after the previous sub-part of the AST has been decoded. Because we make use of an L-attributed grammar, at each step of AST construction, the necessary

inputs are already available from the partial AST at hand. At no point in the decoding process do the neural units need any input from the part of the AST that has not yet been synthesized. This approach to synthesizing is close to the standard inference procedure in sequence-to-sequence models [35, 13]

Given the learned neural units, decoding for the "best" program is an intractable problem, hence we use beam search [37]. Because beam search is only applicable for sequences, we modify it to perform a depth-first traversal on the AST, with the non-terminal nodes that lead to branching in the AST stored in a separate stack.

D Implementation of Baselines

GNN2NAG: We describe the implementation details of GNN2NAG [7], which is one of the baselines in Sec. 5. We expand the AST nodes in the order of a depth-first search. We considered the same six types of edges as Brockschmidt et al. [7], which consists of *Parents*, *Child*, *NextSib*, *NextUse*, *NextToken*, and *InhToSyn*. After building the graph, we propagate the information through a Gated Graph Neural Network (GGNN [22, 7]). We obtain a representation for each node after the GGNN propagation. We then apply an LSTM to go over all the nodes until reaching the node where the goal is to predict the next token. Training is via cross-entropy loss. Note that the biggest difference between our implementation of GNN2NAG and Brockschmidt et al. [7] is the use of an LSTM. Brockschmidt et al. [7] assumes that information about which type of edge is responsible for generating the token is available to the model. However, this information is not available in our setup. Thus, we use an LSTM to iterate over all edges in the GNN to obtain the features for prediction.

Pre-Trained Language Models: We consider 4 types of transformer models—GPTNeo 125M, GPT-Neo 1.3B, CODEGPT, and CODEX [6, 24, 8]. We fine-tuned each of these pre-trained transformers on our Java dataset, except for CODEX, for which we have no access to the pre-trained weights. While our NSG model only takes the headers in the Java class as inputs, for the various transformer models, the input is the entire Java class, including both headers and method bodies. (We found that transformers perform quite poorly if only headers are provided.) During evaluation, transformers are asked to predict the missing method body given the header and the rest of the class.

To fine-tune on our Java dataset, we used the token-level code-completion task provided by CodeXGLUE[3] [24]. During fine-tuning, the transformers are asked to predict the next token in an autoregressive fashion, just like in any language-modeling task. The learning rate is $8e{-}5$ and the batch size is 1 on each GPU. In total, we used 16 GPUs to fine-tune these transformers, which takes about 4 days to complete two epochs on our Java dataset, consisting of 600,000 training instances.

E Generation of Training Data

Now we sketch the process by which our training data is generated. Assume that the task is to generate procedure bodies from start-nonterminal *Prog*, and that we are given a large corpus of Java programs from which to learn the distribution $P(\mathit{Prog}|\mathsf{X})$.

An AG-based compiler is used to produce the training data. For each user-defined method M in the corpus, we create training examples of the form

$$\big((\mathit{Prog}, \mathsf{S}_1^{\text{rhs}}), ..., (S_{i-1}, \mathsf{S}_{i-1}^{\text{rhs}}), (S_i, \mathsf{S}_i^{\text{rhs}}), A(S_i)\!\downarrow, \mathsf{X}\big) \tag{2}$$

where (i) $(\mathit{Prog}, \mathsf{S}_1^{\text{rhs}}), ..., (S_{i-1}, \mathsf{S}_{i-1}^{\text{rhs}})$ is a pre-order listing—from goal nonterminal *Prog* to a particular instance of nonterminal S_i—of the (nonterminal, RHS) choices in M's *Prog* subtree, (ii) $\mathsf{S}_i^{\text{rhs}}$ is the RHS production that occurs at S_i, and (iii) attribute values $A(S_i)\!\downarrow$ are the values at the given instance of S_i. As input-output pairs for a learner, inputs (i) and (iii) produce output (ii).

We compile the program, create its parse tree, and label each node with the values of its attributes (which are evaluated during a left-to-right pass over the tree). For each method M, its subtree is traversed, and a training example is emitted for each node of the subtree.

[3] https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/CodeCompletion-token

Table 7: Percent of Static Checks Passed with 25% Evidence

	GPTNeo125M	GPTNeo1.3B	CodeX	CODEGPT	GNN2NAG	CNG	NSG
No Undeclared Variable Access	90.77%	89.99%	84.55%	89.21%	46.88%	19.78%	**99.32%**
Valid Formal Param Access	NA	NA	NA	NA	25.72%	11.03%	**98.61%**
Valid Class Var Access	NA	NA	NA	NA	14.34%	12.75%	**99.31%**
No Uninitialized Objects	92.35%	91.21%	88.52%	**93.40%**	20.31%	21.56%	93.35%
No Variable Access Error	90.92%	90.11%	84.98%	89.63%	28.10%	17.92%	**99.10%**
Object-Method Compatibility	96.93%	97.05%	96.74%	**98.35%**	21.29%	12.23%	94.87%
Ret Type at Call Site	97.91%	97.66%	**98.47%**	97.98%	22.97%	16.40%	92.53%
Actual Param Type	88.51%	88.61%	90.02%	86.77%	9.22%	16.09%	**93.63%**
Return Stmt Type	82.50%	81.56%	83.75%	83.43%	12.05%	9.51%	**88.94%**
No Type Errors	86.46%	86.14%	88.62%	86.83%	15.98%	13.56%	**91.98%**
Return Stmt Exists	99.58%	99.61%	96.74%	99.56%	93.83%	**99.92%**	96.94%
No Unused Variables	96.51%	96.33%	96.46%	**97.60%**	20.14%	24.29%	91.75%
Percentage of Parsing	98.61%	98.53%	94.95%	97.14%	**100.0%**	**100.0%**	**100.0%**
Pass All Checks	65.26%	62.65%	38.77%	63.12%	16.75%	12.87%	**79.17%**

Table 8: Average Fidelity of Generated Method Bodies with 25% Evidence

	GPTNeo125M	GPTNeo1.3B	CodeX	CODEGPT	CNG	NSG
Set of API Calls	24%	27%	29%	27%	12%	**43%**
Sequences of API Calls	12%	14%	13%	13%	7%	**31%**
Sequences of Program Paths	7%	8%	8%	8%	7%	**28%**
AST Exact Match	7%	8%	8%	8%	1%	**18%**

F Restricting Available Evidence

In our experiments, generation of a particular method is conditioned on available "evidences," which refer to the context surrounding the missing method, in the method's complete class (other method names and method headers, Java Doc comments, class variables, and so on). All of the experiments described thus far simulate the situation where the entire class—except the method to be generated—is visible when it is time to generate the missing method. This simulates the situation where a user is using an automatic programming tool to help generate the very last method in a class, when all other methods and class variables have been defined and are visible.

We can restrict the amount of evidence available to make the task more difficult. When we only make a portion of the evidence available, this simulates the case where a user is using an automatic programming tool to generate a method when the surrounding class is less complete. When we use "$x\%$ evidence" for a task, each piece of evidence in the surrounding code is selected and available to the automatic programming tool with $x\%$ probability. In Table 7 and Table 8, we show results obtained when we repeat the experiments from earlier in the paper, but this time using 25% evidence while Table 9 and Table 10 show the result for 50% evidence.

G Next-Token Prediction

Our NSG implementation uses a relatively weak language model (based on LSTMs as opposed to more modern transformers) but augments them with a static analysis. We have shown that the resulting NSG is good at "long-horizon" tasks such as semantic consistency (compared to the baselines tested) and at generating methods that have high fidelity to the original, "correct" method. But it is reasonable to ask: how does the NSG compare to the baselines at "short-horizon" tasks? To measure this, for each symbol S that is expanded to form the body of a test method, we compute (i) the actual left-context sequence of the test method (up to but not including the RHS sequence chosen for S) as the value of SymSoFar, (ii) $A(S)\downarrow$ (in the case of the NSG), and (iii) Z. We then use these values to ask the NSG to predict the next RHS. If the predicted RHS matched the observed RHS, the model was scored as "correct." We recorded the percentage of correct predictions for terminal RHS symbols (such as API calls or types) for each test program.

We also performed next-token prediction using the NSG and three of the baseline models. Note that it is non-trivial to classify CODEGPT's output into different terminal symbols, so we only report the overall RHS symbols' correctness. The results show that two of the baselines (CODEGPT and GNN2NAG) are very accurate, and demonstrate better performance than the NSG on this task.

Table 9: Percent of Static Checks Passed with 50% Evidence

	GPTNeo125M	GPTNeo1.3B	CodeX	CODEGPT	GNN2NAG	CNG	NSG
No Undeclared Variable Access	89.87%	90.36%	88.62%	90.34%	47.17%	17.79%	**99.86%**
Valid Formal Param Access	NA	NA	NA	NA	25.50%	8.58%	**99.83%**
Valid Class Var Access	NA	NA	NA	NA	14.96%	11.57%	**99.78%**
No Uninitialized Objects	93.90%	91.73%	90.82%	94.37%	20.01%	21.68%	**97.30%**
No Variable Access Error	90.36%	90.51%	88.86%	91.32%	28.43%	17.34%	**99.84%**
Object-Method Compatibility	**98.36%**	98.09%	98.35%	97.84%	21.39%	10.11%	96.42%
Ret Type at Call Site	97.38%	98.01%	**98.53%**	97.83%	23.45%	14.82%	97.22%
Actual Param Type	87.03%	86.36%	92.28%	88.71%	9.24%	14.35%	**96.74%**
Return Stmt Type	84.05%	85.09%	88.13%	85.23%	12.07%	7.66%	**92.15%**
No Type Errors	87.25%	88.13%	91.42%	88.10%	16.04%	11.45%	**96.22%**
Return Stmt Exists	99.61%	99.80%	98.44%	**99.57%**	93.87%	98.71%	97.47%
No Unused Variables	96.42%	96.46%	96.82%	**97.64%**	20.55%	18.50%	94.20%
Percentage of Parsing	98.18%	98.13%	94.69%	97.08%	**100.0%**	**100.0%**	**100.0%**
Pass All Checks	65.26%	64.88%	47.49%	67.73%	16.92%	24.28%	**86.00%**

Table 10: Average Fidelity of Generated Method Bodies with 50% Evidence

	GPTNeo125M	GPTNeo1.3B	CodeX	CODEGPT	CNG	NSG
Set of API Calls	32%	37%	36%	36%	12%	**50%**
Sequences of API Calls	17%	20%	16%	19%	7%	**39%**
Sequences of Program Paths	13%	10%	10%	14%	7%	**36%**
AST Exact Match	13%	10%	10%	14%	1%	**21%**

These results are in-keeping with our assertion that the baselines are useful mostly for short-horizon code-generation tasks. However, they struggle with long-horizon tasks, such as the CPG task of generating an entire Java method body. The results—together our earlier CPG results—also show that even though the NSG has reduced accuracy in a short-horizon task, it is still able to generate semantically accurate programs on the CPG task.

H Application to Novel Semantic Checks

The NSG approach can generate semantically accurate programs given context. At its core, an NSG relies on the various semantic properties (i.e., attributes) on which it is trained. We would like to understand the influence of these semantic properties in the generated program, and explore the possibility that training on such a set of attributes can automatically allow for high accuracy with respect to additional semantic checks for which specific attributes were not explicitly provided during training. To study this question, we performed an ablation study in which we trained an NSG with a subset of the relevant attributes, but evaluated the generated programs on all properties.

We trained an NSG without the *attrOut.retStmtGenerated* and *methodRetType* attributes, as defined in Section K.2. With 50% of the evidence available, we see that the resulting model suffers in terms of accuracy. The "Return Stmt Type" accuracy falls from 92.15% to 77.45% whereas the "Return Stmt Exists" accuracy falls from 97.47% to 95.68%. That said, note that the resulting "Return Stmt Type" accuracy is still a big improvement over the vanilla CNG model (with no attributes), which is correct only 9.51% of the time.

This suggests that the NSG has learned type-safe program generation from other semantic properties, most notably the *symTab* attribute, which carries type information about the various objects that are currently in scope. This further suggests that providing a small core of key attributes may be enough to greatly increase the accuracy of code generation.

I Robustness Incomplete Analysis

In this section, we analyze a situation where the static analyzer fails to accurately resolve different attributes during the synthesis process. We simulate three situations in which the static analyzer might fail.

In the first scenario, we emulate a situation where the compiler is unable to resolve the correct return-type information from the missing method that the user has asked the NSG to synthesize. This

Table 11: Next-Token Prediction Accuracy

	Percentage of Evidence Available							
	50%				100%			
	NSG	CODEGPT	GNN2NAG	CNG	NSG	CODEGPT	GNN2NAG	CNG
API Calls	62.42%	NA	**80.24%**	49.05%	75.94%	NA	**80.77%**	59.73%
Object Initialization Call	59.64%	NA	**97.65%**	49.12%	66.66%	NA	**97.94%**	87.90%
Types	**61.11%**	NA	**85.78%**	50.28%	70.33%	NA	**86.21%**	54.44%
Variable Access	**92.26%**	NA	92.11%	50.28%	92.44%	NA	**92.94%**	52.85%
All Terminal RHS Symbols	73.41%	**88%**	80.83%	51.22%	73.99%	**89%**	81.1%	54.32%

Real Code	CodeGPT	NSG

`public String reader()`
`{`
`  StringBuffer stringBuffer`
`  = new StringBuffer();`
`  String line;`
`  while ((line =`
`  bReader.readLine() ~= null) {`
`  stringBuffer.append(line);`
`  stringBuffer.append("\n");};`
`  return stringBuffer.toString();`
`}`
	`public String reader()`
	`{`
	`  StringBuffer buffer=`
	`  new StringBuffer();`
	`  buffer.append("\n");`
	`  return buffer.toString();`
	`}`
		`public String reader()`
		`{`
		`  java.lang.String var_9;`
		`  try{`
		`  var_9=field_5.readLine();`
		`  }`
		`  catch(IOException var_8) {`
		`  var_8.printStackTrace();`
		`  }`
		`  return var_9;`
		`}`

Table 12: Reader example for analyzing the BLEU-score metric.

results in a default `null` value being passed around for the attribute *methodRetType*. We find that this reduces the overall accuracy for the attribute "Return Stmt Type" from 90.97% to 77.28%. This does not seem to impact other static checks, however.

In the second scenario, consider the case where the compiler is unable to resolve the API return-type attribute *retType*. This reduces the accuracy of the "Return type at call site" check from 98.01% to 18.16%. It also results in a decrease in "No undeclared-variable access" and "Valid formal-param access" to 72.48% and 67.23%, respectively. This is is a huge decrease from 99.82% and 99.55% accuracy that these semantic checks had for the base model where the attribute *retType* can be resolved correctly. The fidelity metrics are also impacted, where the "AST exact match" metric drops from 26% to 10%. This is because the *retType* attribute is used in many portions of our attribute grammar, on which the trained program generator is being conditioned on. An incorrect resolution of such attribute had led to deterioration of the overall model performance.

In the final scenario, we only break the static analyzer's capability to resolve the unused-variable-check attribute *attrOut.isUsed*. For this scenario, we see that only the one semantic check "No unused variables" out of all the semantic checks considered is impacted. Here the accuracy for this check drops from 93.84% to 91.10%. All other metrics have negligible changes.

Rather unsurprisingly, the results suggest that NSG relies heavily on the static analyzer, and that some attributes influence the result much more than others. It is also critical to have a static analyzer that performs accurately during inference time, to avoid any model performance degradation.

J BLEU-Score Analysis

As described in the main body of the paper, BiLingual Evaluation Understudy or BLEU score is "problematic in the code-generation setting. First, the BLEU score is not invariant to variable renamings, which means that a nonsensical program that uses commonplace variable names can get an artificially high BLEU score. Second, programs are structured objects in which some tokens indicate control flow, and some indicate data flow. The BLEU score does not take this structure into account." We use one of the examples in Table 4 of the paper ("reading from a file") to illustrate this point and show the real code and outputs from CodeGPT and NSG in Table 12.

The NSG output is clearly better. However, the CodeGPT output gets a higher BLEU score because it uses variable names that superficially match the ground truth. Specifically, the BLEU score of the CodeGPT output is 25.11 and the BLEU score of the NSG output is 19.07. This situation arose often in our experiments, which is why we have used alternative program-equivalence metrics to judge performance of generated programs, as defined in Section 5.

K Grammar

The Neural Attribute Grammar (NSG) model learns to synthesize real-life Java programs while learning over production rules of an attribute grammar. In this section, we present the comprehensive set of production rules considered, along with the attributes used. We first present the context-free grammar in Appendix K.1, and then decorate it with attributes in Appendix K.2. The productions in **a-c** deal with expansion of all the non-terminal symbols in the grammar: rules in **a** mainly expand to one line of code in a Java method body; rules in **b** are their corresponding expansions; and rules in **c** deal with control-flow operations inside the grammar. Rules in **d** generate terminal symbols inside the grammar. We show the flow of attributes *symTab* and *methodRetType* in the AST in Appendix K.2. The rest of the attributes are passed inside *attrIn* and *attrOut*, namely *isInitialized*, *isUsed*, *retStmtGenerated* and *itrVec*.

K.1 Context Free Grammar

a1. *Start* : *Stmt*
a2. *Stmt* : *Stmt* ; *Stmt* | ϵ
a3. *Stmt* : *Decl*
a4. *Stmt* : *ObjInit*
a5. *Stmt* : *Invoke*
a6. *Stmt* : *Return*

b1. *Decl* : *Type Var*
b2. *ObjInit* : *Type Var* **= new** *Type ArgList*
b3. *Invoke* : *Var* = *Var Call InvokeMore*
b4. *InvokeMore* : *Call InvokeMore* | ϵ
b5. *Call* : *Api ArgList*
b6. *ArgList* : *Var ArgList* | ϵ
b7. *Return* : **return** *Var*

c1. *Stmt* : *Branch* | *Loop* | *Except*
c2. *Branch* : **if** *Cond* **then** *Stmt* **else** *Stmt*
c3. *Loop* : **while** *Cond* **then** *Stmt*
c4. *Except* : **try** *Stmt Catch*
c5. *Catch* : **catch**(*Type*) *Stmt*; *Catch* | ϵ
c6. *Cond* : *Call*

d1. *Api* : **JAVA_API_CALL**
d2. *Api* : **INTERNAL_METHOD_CALL**
d3. *Type* : **JAVA_TYPE**
d4. *Var* : **VAR_ID**

K.2 Attribute Grammar

a0. Initialization of inherited attributes of *Start*:
$[$ *Start*.symTab $\downarrow$:=
$\{ in\_param\_1 \mapsto type\_in\_param\_1,$
$\ldots$
$in\_param\_n \mapsto type\_in\_param\_n,$
$field\_1 \mapsto type\_field\_1,$
$\ldots$
$field\_m \mapsto type\_field\_m \}$
Start.attrIn.itrVec $\downarrow$:= (false, false);
Start.attrIn.retStmtGenerated $\downarrow$:= false;
Start.attrIn.isInitialized $\downarrow$:= ϕ;
Start.attrIn.isUsed $\downarrow$:= ϕ;
Start.methodRetType $\downarrow$:= $METHOD\_RET\_TYPE;]$

a1. *Start* : *Stmt* ;
$[$ *Stmt*.attrIn $\downarrow$:= *Start*.attrIn $\downarrow$;
Stmt.methodRetType $\downarrow$:= *Start*.methodRetType $\downarrow$;
Stmt.symTab $\downarrow$:= *Start*.symTab $\downarrow$;
Start.symTabOut $\uparrow$:= *Stmt*.symTabOut $\uparrow$;
Start.attrOut $\uparrow$:= *Stmt*.attrOut $\uparrow$;
Start.valid $\uparrow$:= *Stmt*.valid $\uparrow$;$]$

a2a. *Stmt*\$0 : *Stmt*\$1 ; *Stmt*\$2
$[$ *Stmt*\$1.symTab $\downarrow$:= *Stmt*\$0.symTab $\downarrow$;
Stmt\$2.symTab $\downarrow$:= *Stmt*\$1.symTabOut $\uparrow$;
Stmt\$0.symTabOut $\uparrow$:= *Stmt*\$2.symTabOut $\uparrow$;
Stmt\$1.attrIn $\downarrow$:= *Stmt*\$0.attrIn $\downarrow$;
Stmt\$2.attrIn $\downarrow$:= *Stmt*\$1.attrOut $\uparrow$;
Stmt\$0.attrOut $\uparrow$:= *Stmt*\$2.attrOut $\uparrow$;
Stmt\$1.methodRetType $\downarrow$:= *Stmt*\$0.methodRetType $\downarrow$;
Stmt\$2.methodRetType $\downarrow$:= *Stmt*\$0.methodRetType $\downarrow$;
Stmt\$0.valid $\uparrow$:= *Stmt*\$1.valid $\uparrow \wedge$ *Stmt*\$2.valid $\uparrow$;$]$

a2b. *Stmt* : ϵ
$[$ *Stmt*.symTabOut $\uparrow$:= {};
Stmt.attrOut.itrVec $\uparrow$:= (false, false);
Stmt.valid $\uparrow$:= true;$]$

a3. *Stmt* : *Decl*
$[$ *Decl*.symTab $\downarrow$:= *Stmt*.symTab $\downarrow$;
Stmt.symTabOut $\uparrow$:= *Stmt*.symTab $\downarrow$ + *Decl*.symTabOut $\uparrow$;
Decl.attrIn $\downarrow$:= *Stmt*.attrIn $\downarrow$;
Stmt.attrOut $\uparrow$:= *Stmt*.attrIn $\downarrow$ + *Decl*.attrOut $\uparrow$;
Decl.methodRetType $\downarrow$:= *Stmt*.methodRetType $\downarrow$;
Stmt.valid $\uparrow$:= *Decl*.valid $\uparrow$;$]$

a4. *Stmt* : *ObjInit*
$[$ *ObjInit*.symTab $\downarrow$:= *Stmt*.symTab $\downarrow$;
Stmt.symTabOut $\uparrow$:= *Stmt*.symTab $\downarrow$
+ *ObjInit*.symTabOut $\uparrow$;
ObjInit.attrIn $\downarrow$:= *Stmt*.attrIn $\downarrow$;
ObjInit.methodRetType $\downarrow$:= *Stmt*.methodRetType $\downarrow$;
Stmt.attrOut $\uparrow$:= *Stmt*.attrIn $\downarrow$ + *ObjInit*.attrOut $\uparrow$;
Stmt.valid $\uparrow$:= *ObjInit*.valid $\uparrow$ $]$

a5. *Stmt* : *Invoke*
$[$ *Invoke*.symTab $\downarrow$:= *Stmt*.symTab $\downarrow$;
Stmt.symTabOut $\uparrow$:= *Stmt*.symTab $\downarrow$ +*Invoke*.symTabOut $\uparrow$;
Invoke.attrIn $\downarrow$:= *Stmt*.attrIn $\downarrow$;
Stmt.attrOut $\uparrow$:= *Stmt*.attrOut $\downarrow$ +*Invoke*.attrOut $\uparrow$;
Invoke.methodRetType $\downarrow$:= *Stmt*.methodRetType $\downarrow$;
Stmt.valid $\uparrow$:= *Invoke*.valid $\uparrow$;$]$

a6. *Stmt* : *Return*
$[$ *Return*.symTab $\downarrow$:= *Stmt*.symTab $\downarrow$;
Stmt.symTabOut $\uparrow$:=
Stmt.symTab $\downarrow$ + *Return*.symTabOut $\uparrow$;
Invoke.attrIn $\downarrow$:= *Stmt*.attrIn $\downarrow$;
Stmt.attrOut $\uparrow$:= *Stmt*.attrIn $\downarrow$ + *Invoke*.attrOut $\uparrow$;$]$
Return.methodRetType $\downarrow$:= *Stmt*.methodRetType $\downarrow$;
Stmt.valid $\uparrow$:= *Return*.valid $\uparrow$;$]$

b1. *Decl* : *Type Var*
$[$*Decl*.symTabOut $\uparrow$:= {*Var*.id : *Type*.name};
Decl.attrOut.isUsed[*Var*] $\uparrow$:= false;
Decl.attrOut.isInitialized[*Var*] $\uparrow$:= false;
Decl.valid $\uparrow$:= true$]$

b2. *ObjInit* : *Type*\$0 *Var* **= new** *Type*\$1 *ArgList*
$[$ *ArgList*.symTab $\downarrow$:= *ObjInit*.symTab $\downarrow$;
ObjInit.symtabOut $\uparrow$:= $\{$*Var*.id : *Type*.name$\}$;
ArgList.typeList $\downarrow$:= *Type*.params $\uparrow$;
ObjInit.attrOut.isInitialized[*Var*] $\uparrow$:= true;
ObjInit.attrOut.isUsed[*Var*] $\uparrow$:= false;
ObjInit.valid $\uparrow$:= *ArgList*.valid $\uparrow$;
$\wedge$ *Type*\$0.name $\uparrow$:= *Type*\$1.name $\uparrow$;$]$

b3. *Invoke* : *Var*\$0 = *Var*\$1 *Call InvokeMore*
$[$ *InvokeMore*.symTab $\downarrow$:= *Invoke*.symTab $\downarrow$;
InvokeMore.exprType $\downarrow$:= *Call*.retType $\uparrow$;
Call.attrIn $\downarrow$:= *Invoke*\$0.attrIn $\downarrow$;
InvokeMore.attrIn $\downarrow$:= *Call*.attrOut $\uparrow$;
Invoke.attrOut.isUsed[*Var*\$0] $\uparrow$:= true;
Invoke.attrOut.isUsed[*Var*\$1] $\uparrow$:= true;
Invoke.attrOut $\uparrow$:= *InvokeMore*.attrOut $\uparrow$;
Invoke.valid $\uparrow$:= *InvokeMore*.valid $\uparrow$
$\wedge$ (*InvokeMore*.retType $\uparrow$==
Invoke.symTab $\downarrow$ [*Var*\$0.id $\uparrow$])
$\wedge$ *Call*.exprType $\uparrow$ ==
Invoke.symTab $\downarrow$ [*Var*\$1.id $\uparrow$];$]$

b4a. *InvokeMore*\$0 : *Call InvokeMore*\$1
$[$ *InvokeMore*\$1.symTab $\downarrow$:= *InvokeMore*\$0.symTab $\downarrow$;
InvokeMore\$1.exprType $\downarrow$:= *Call*.returnType $\uparrow$;
Call.symTab $\downarrow$:= *InvokeMore*\$0.symTab $\downarrow$;
InvokeMore\$0.retType $\uparrow$:= *InvokeMore*\$1.retType $\uparrow$;
Call.attrIn $\downarrow$:= *InvokeMore*\$0.attrIn $\downarrow$;
InvokeMore\$1.attrIn $\downarrow$:= *Call*.attrOut $\uparrow$;
InvokeMoreOut\$0.attrIn $\uparrow$:= *InvokeMoreOut*\$1.attrIn $\uparrow$;
InvokeMore\$0.valid $\uparrow$:= *Call*.valid $\uparrow$
$\wedge$ *InvokeMore*\$1.valid $\uparrow$;
$\wedge$ *Call*.exprType $\uparrow$:= *InvokeMore*\$1.exprType $\downarrow$;$]$

b4b. *InvokeMore* : ϵ
$[$ *InvokeMore*.retType $\uparrow$:= *InvokeMore*.exprType $\downarrow$;
InvokeMore.attrIn.itrVec $\uparrow$= (false, false);
InvokeMore.valid $\uparrow$:= true;$]$

b5 *Call* : *Api ArgList*
$[$ *ArgList*.symTab $\downarrow$:= *Call*.symTab $\downarrow$;
ArgList.typeList $\downarrow$:= *Api*.params $\uparrow$;
Call.retType $\uparrow$:= *Api*.retType $\uparrow$]
Api.attrIn $\downarrow$:= *Call*.attrIn $\downarrow$;
Call.attrOut $\uparrow$:= *Api*.attrOut $\uparrow$;
Call.exprType $\uparrow$:= *Api*.exprType $\uparrow$);
Call.valid $\uparrow$:= *ArgList*.valid $\uparrow$ $]$

b6a. *ArgList*\$0 : *Var ArgList*\$1
$[$ *ArgList*\$1.symTab $\downarrow$:= *ArgList*\$0.symTab $\downarrow$;
ArgList\$1.typeList $\downarrow$:= *ArgList*\$0.typeList[1 :] $\downarrow$;
ArgList\$1.attrOut.isUsed[*Var*] $\uparrow$:= true;
ArgList\$0.valid $\uparrow$:= *ArgList*\$1.valid $\uparrow$
$\wedge$ (*ArgList*\$0.symTab $\downarrow$ [*Var*.id $\uparrow$]
== *ArgList*\$0.typeList[0] $\downarrow$);$]$

b6b. *ArgList* : ϵ
$[$ *ArgList*.valid $\uparrow$:= *ArgList*.typeList.$isEmpty()$ $\uparrow$;$]$

b7. *Return* : **return** *Var*
$[$ *Return*.attrOut.retStmtGenerated $\uparrow$ :: true
Return.valid $\uparrow$:= *Return*.methodRetType $\downarrow$==
Return.symTab $\downarrow$ [*Var*.id $\uparrow$]$]$;

c1.a. *Stmt* : *Branch*
$[\ \mathit{Branch}.\text{symTab}\downarrow\ :=\ \mathit{Stmt}.\text{symTab}\downarrow;$
$\mathit{Stmt}.\text{valid}\uparrow\ :=\ \mathit{Branch}.\text{valid}\uparrow;$
$\mathit{Branch}.\text{attrIn}\downarrow\ :=\ \mathit{Stmt}.\text{attrIn}\downarrow;$
$\mathit{Stmt}.\text{attrOut}\uparrow\ :=\ \mathit{Branch}.\text{attrOut}\uparrow;\ ]$

c1.b. *Stmt* : *Loop*
$[\ \mathit{Loop}.\text{symTab}\downarrow\ :=\ \mathit{Stmt}.\text{symTab}\downarrow;$
$\mathit{Stmt}.\text{valid}\uparrow\ :=\ \mathit{Loop}.\text{valid}\uparrow;$
$\mathit{Loop}.\text{attrIn}\downarrow\ :=\ \mathit{Stmt}.\text{attrIn}\downarrow;$
$\mathit{Stmt}.\text{attrOut}\uparrow\ :=\ \mathit{Loop}.\text{attrOut}\uparrow;\ ]$

c1.c. *Stmt* : *Except*
$[\ \mathit{Except}.\text{symTab}\downarrow\ :=\ \mathit{Stmt}.\text{symTab}\downarrow;$
$\mathit{Stmt}.\text{valid}\uparrow\ :=\ \mathit{Except}.\text{valid}\uparrow;$
$\mathit{Except}.\text{attrIn}\downarrow\ :=\ \mathit{Stmt}.\text{attrIn}\downarrow;$
$\mathit{Stmt}.\text{attrOut}\uparrow\ :=\ \mathit{Except}.\text{attrOut}\uparrow;\ ]$

c2. *Branch* : **if** *Cond* **then** *Stmt*\$1 **else** *Stmt*\$2
$[\ \mathit{Cond}.\text{symTab}\downarrow\ :=\ \mathit{Stmt}.\text{symTab}\downarrow;$
$\mathit{Stmt}\$1.\text{symTab}\downarrow\ :=\ \mathit{Cond}.\text{symtabOut}\uparrow;$
$\mathit{Stmt}\$2.\text{symTab}\downarrow\ :=\ \mathit{Cond}.\text{symtabOut}\uparrow;$
$\mathit{Branch}.\text{valid}\uparrow\ :=\ \mathit{Cond}.\text{valid}\uparrow\ \wedge\ \mathit{Stmt}\$1.\text{valid}\uparrow$
$\wedge\ \mathit{Stmt}\$2.\text{valid}\uparrow;$
$\mathit{Cond}.\text{attrIn}\downarrow\ :=\ \mathit{Branch}.\text{attrIn}\downarrow;$
$\mathit{Stmt}\$1.\text{attrIn}\downarrow\ :=\ \mathit{Cond}.\text{attrIn}\downarrow;$
$\mathit{Stmt}\$2.\text{attrIn}\downarrow\ :=\ \mathit{Cond}.\text{attrIn}\downarrow;$
$\mathit{Branch}.\text{attrOut}\uparrow\ :=\ \mathit{Branch}\$1.\text{attrIn}\downarrow;\ ]$

c3. *Loop* : **while** *Cond* **then** *Stmt*
$[\ \mathit{Cond}.\text{symTab}\downarrow\ :=\ \mathit{Stmt}.\text{symTab}\downarrow;$
$\mathit{Stmt}.\text{symTab}\downarrow\ :=\ \mathit{Cond}.\text{symTabOut}\uparrow;$
$\mathit{Loop}.\text{valid}\uparrow\ :=\ \mathit{Cond}.\text{valid}\uparrow\ \wedge\ \mathit{Stmt}.\text{valid}\uparrow;$
$\mathit{Cond}.\text{attrIn}\downarrow\ :=\ \mathit{Loop}.\text{attrIn}\downarrow;$
$\mathit{Stmt}.\text{attrIn}\downarrow\ :=\ \mathit{Cond}.\text{attrOut}\uparrow;$
$\mathit{Loop}.\text{attrOut}\uparrow\ :=\ \mathit{Loop}.\text{attrIn}\downarrow;\ ]$

c4. *Except* : **try** *Stmt Catch*
$[\ \mathit{Stmt}.\text{symTab}\downarrow\ :=\ \mathit{Except}.\text{symTab}\downarrow;$
$\mathit{Catch}.\text{symTab}\downarrow\ :=\ \mathit{Stmt}.\text{symTabOut}\uparrow;$
$\mathit{Except}.\text{valid}\uparrow\ :=\ \mathit{Stmt}.\text{valid}\uparrow\ \wedge\ \mathit{Catch}.\text{valid}\uparrow;$
$\mathit{Stmt}.\text{attrIn}\downarrow\ :=\ \mathit{Except}.\text{attrIn}\downarrow;$
$\mathit{Catch}.\text{attrIn}\downarrow\ :=\ \mathit{Stmt}.\text{attrIn}\downarrow;$
$\mathit{Except}.\text{attrOut}\uparrow\ :=\ \mathit{Except}.\text{attrIn}\downarrow;\ ]$

c5a. $\mathit{Catch}\$0$: **catch**(Type) *Stmt*; $\mathit{Catch}\$1$
$[\ \mathit{Catch}\$1.\text{symTab}\downarrow\ :=\ \mathit{Catch}\$0.\text{symTab}\downarrow;$
$\mathit{Stmt}.\text{symTab}\downarrow\ :=\ \mathit{Catch}\$0.\text{symTab}\downarrow;$
$\mathit{Stmt}.\text{attrIn}\downarrow\ :=\ \mathit{Catch}\$0.\text{attrIn}\downarrow;$
$\mathit{Catch}\$1.\text{attrIn}\downarrow\ :=\ \mathit{Stmt}.\text{attrIn}\downarrow;$
$\mathit{Catch}\$0.\text{attrOut}\uparrow\ :=\ \mathit{Catch}\$1.\text{attrOut}\uparrow;$
$\mathit{Catch}\$0.\text{valid}\uparrow\ :=\ \mathit{Stmt}.\text{valid}\uparrow\ \wedge\mathit{Catch}\$1.\text{valid}\uparrow;\ ]$

c5b. *Catch* : ϵ
$[\ \mathit{Catch}.\text{valid}\uparrow\ :=\ \text{true};\ \mathit{Catch}.\text{attrOut}\uparrow\ :=\ \phi\ ]$

c6. *Cond* : *Call*
$[\ \mathit{Call}.\text{symTab}\downarrow\ :=\ \mathit{Cond}.\text{symTab}\downarrow;$
$\mathit{Cond}.\text{valid}\uparrow\ :=\ \mathit{Call}.\text{valid}\uparrow;$
$\mathit{Call}.\text{attrIn}\downarrow\ :=\ \mathit{Cond}.\text{attrIn}\downarrow;$
$\mathit{Cond}.\text{attrOut}\uparrow\ :=\ \mathit{Call}.\text{attrOut}\uparrow;\ ]$

d1. *Api* : **JAVA_API_CALL**
$\big[$ *Api*.name $\uparrow$ $:=$ $NAME$;
Api.params $\uparrow$ $:=$ $FORMAL\_PARAM\_LIST$;
Api.exprType $\uparrow$ $:=$ $TYPE$;
Api.retType $\uparrow$ $:=$ RET_TYPE;
if(*Api*.name $==$ "hasNext")
  Api.attrOut.itrVec $\uparrow$ $:=$ (true, false);
else if(*Api*.name $==$ "next")
  Api.attrOut.itrVec[1] $\uparrow$ $:=$ true;$\big]$

d2. *Api* : **INTERNAL_METHOD_CALL**
$\big[$ *Api*.name $\uparrow$ $:=$ $NAME$;
Api.params $\uparrow$ $:=$ $FORMAL\_PARAM\_LIST$;
Api.exprType $\uparrow$ $:=$ ϵ;
Api.retType $\uparrow$ $:=$ RET_TYPE;
Api.attrOut.itrVec $\uparrow$ $:=$ *Api*.attrIn.itrVec $\downarrow$;$\big]$

d3. *Type* : **JAVA_TYPE**
$\big[$ *Type*.name $\uparrow$ $:=$ $NAME$
Type.params $\uparrow$ $:=$ $FORMAL\_PARAM\_LIST$;$\big]$

d4. *Var* : **VAR_ID**
$\big[$ *Var*.id $\uparrow$ $:=$ $ID\_NUMBER\big]$